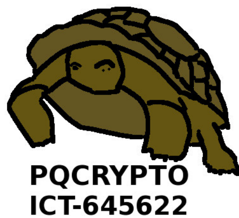

PQCRYPTO

Post-Quantum Cryptography for Long-Term Security

Project number: Horizon 2020 ICT-645622

D2.5

Internet: Integration

Due date of deliverable: 1. March 2018
Actual submission date: 12. April 2018

WP contributing to the deliverable: WP2

Start date of project: 1. March 2015 Duration: 3 years

Coordinator:
Technische Universiteit Eindhoven
Email: coordinator@pqcrypto.eu.org
www.pqcrypto.eu.org

Revision 1.00

Project co-funded by the European Commission within Horizon 2020		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

Internet: Integration

Daniel J. Bernstein

12. April 2018
Revision 1.00

The work described in this report has in part been supported by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Abstract

This document describes Internet integration of `libpqcrypto`, WP2's software library for post-quantum cryptography, including a successful demo of high-speed high-security post-quantum Internet communication.

Keywords: protocols, Internet, post-quantum cryptography

Contents

1 Introduction 3

2 Security warnings 3

3 An easy example: integrity for software updates 3
- 3.1 Current practice: pre-quantum signatures . . . 3
- 3.2 Using libpqcrypto for post-quantum signatures . . . 4
- 3.3 Combining pre-quantum and post-quantum systems . . . 5
- 3.4 The importance of auditing . . . 6
- 3.5 Alternatives . . . 6

4 A more challenging example: confidentiality, integrity, and availability for web browsing 7

5 Non-cryptographic Internet communication 8
- 5.1 IP: Internet Protocol . . . 8
- 5.2 UDP: User Datagram Protocol . . . 8
- 5.3 DNS: Domain Name System . . . 9
- 5.4 TCP: Transmission Control Protocol . . . 9
- 5.5 HTTP: Hypertext Transfer Protocol . . . 10
- 5.6 Caching . . . 11
- 5.7 More protocols . . . 11

6 How is cryptography triggered? 12
- 6.1 Opportunistic encryption: e.g., STARTTLS . . . 12
- 6.2 Trust on first use: e.g., HPKP . . . 12
- 6.3 Server+client-side configuration per server+protocol: e.g., HTTPS . . . 12
- 6.4 Server+client-side configuration per server: e.g., VPNs . . . 13
- 6.5 Server-side configuration per server: e.g., MinimaLT . . . 13

7 Which protocol versions can the client and server choose? 14
- 7.1 Reasons for upgrading cryptographic choices inside protocols . . . 14
- 7.2 Single-version client, single-version server . . . 14
- 7.3 Multiple-version client, single-version server . . . 15
- 7.4 Single-version client, multiple-version server . . . 16
- 7.5 Multiple-version client, multiple-version server . . . 16

8 How does the server's public key authenticate the server? 17
- 8.1 Public-key encryption . . . 17
- 8.2 Public-key signatures . . . 17

9 How long are secret keys stored? 18
- 9.1 Forever . . . 18
- 9.2 Until the end of a session . . . 18
- 9.3 Until the next packet is received . . . 18
- 9.4 Until two minutes later . . . 19

10 How does the client resume a session? 21
10.1 No resumption 21
10.2 Refreshing the session key 21
10.3 Eliminating server storage 21

11 PQConnect 22

1 Introduction

PQCRYPTO's `libpqcrypto` software library includes 77 cryptographic systems from 19 of the 22 PQCRYPTO submissions to NIST's ongoing post-quantum standardization project. This document reports a successful demo by PQCRYPTO using `libpqcrypto` for high-speed high-security post-quantum Internet communication. More broadly, this document analyzes options for integration of `libpqcrypto` into Internet communication. A detailed description of `libpqcrypto` per se is in a separate document, D2.4.

WP2's original goal was to add post-quantum protection to aspects of the Internet that merely have pre-quantum protection today. However, WP2 has identified a "stretch goal" of adding post-quantum protection to aspects of the Internet that have *no* protection today. For example, HTTPS and other protocols built on top of Transport Layer Security (TLS) are vulnerable to low-cost denial-of-service attacks; the analysis in this document includes ways to make denial-of-service attacks more difficult.

The new demo reported here has an important advantage over the successful "CECPQ1" experiment that ran from July 2016 through November 2016 between the Chrome browser and Google's servers, using PQCRYPTO's "New Hope" cryptographic design and PQCRYPTO's cryptographic software. The CECPQ1 experiment aimed for "transitional security", using a post-quantum key exchange to protect confidentiality of communication against quantum computers that are built *after* the communication takes place. However, CECPQ1 did not attempt to provide post-quantum authentication. A man-in-the-middle attacker equipped with a quantum computer can break the integrity and confidentiality of all subsequent use of CECPQ1; so CECPQ1 must be replaced by something more secure before quantum computers appear. The new demo aims for full post-quantum security, encrypting and authenticating communication to protect both confidentiality and integrity against quantum computers, even if the quantum computers are built *before* the communication takes place.

2 Security warnings

See D2.4 for security warnings relevant to `libpqcrypto`. Even if `libpqcrypto` is secure, the higher-level networking protocols and implementations considered in this document need further auditing and verification. Furthermore, even if these protocols achieve their security goals, much more work is required on advanced security goals such as location privacy.

3 An easy example: integrity for software updates

3.1 Current practice: pre-quantum signatures

When your computer downloads a new version of its operating system (OS), it checks a signature on the download from the OS author. This is a critical use of cryptography: if these signatures were not checked then attackers would be able to easily insert malware into your computer by sending you a fake update to the OS.

For example, updates to the security-oriented OpenBSD operating system are signed using a state-of-the-art elliptic-curve signature system, Ed25519. This signature system has also been deployed in many other applications (see generally https://ianix.com/pub/ed25519-deployment.html), and has been standardized for various IETF protocols.

Ed25519 was designed in 2011 by Bernstein (now PQCRYPTO WP2 leader), Duif (at the time a student at TU/e), Lange (now PQCRYPTO coordinator), Schwabe (now PQCRYPTO WP1 co-leader), and Yang (Academia Sinica, now part of PQCRYPTO), but this does not mean that it resists quantum computers. On the contrary: Ed25519, like the rest of elliptic-curve cryptography, will be broken by Shor's algorithm.

This raises the question of how to protect software updates against an attacker armed with a quantum computer. It is critical to fully deploy a high-security post-quantum solution *before* any quantum computers become available to attackers. Furthermore, for two reasons, one cannot wait until the last moment to make an update available that switches to post-quantum cryptography for all future updates. The first reason is that the moment is not known: one cannot expect a serious attacker to announce "Now I have a quantum computer. HO-HO-HO." The second reason is that Internet devices often take years to be upgraded,[1] so the post-quantum update needs to be *available* years before it needs to be *deployed*.

3.2 Using `libpqcrypto` for post-quantum signatures

The obvious way to protect the long-term integrity of OS updates is to take each deployed pre-quantum signature system P and replace it with a post-quantum signature Q. This is an easy example of upgrading from pre-quantum security to post-quantum security:

- The OS updates being signed are quite large, even if the updates are split into individually signed "packages". Signature overhead is thus relatively small. Switching from 64-byte Ed25519 signatures to, e.g., 49216-byte `sphincsf256sha256` signatures might sound like a dramatic size increase; but in the Debian OS, which is designed for frequent updates, the average size of a single package is 1.2 megabytes. If signature overhead were a concern then the OS could further reduce the overhead by signing commonly used bundles of packages.

- Similarly, switching from under 200000 cycles for Ed25519 signature verification to millions of CPU cycles for `sphincsf256sha256` signature verification might sound like a dramatic slowdown, but either way millions of CPU cycles are spent hashing the package being verified, and much more time is spent installing the package.

- The bigger picture is that computers spend almost all of their network traffic and CPU time on things other than OS updates.

Subsequent sections of this document switch to a more difficult example, namely protecting web traffic, but in this section there are no serious performance concerns.

The usual advertisement for hash-based signatures, such as `sphincsf256sha256`, is that forging a signature—given as input the public key and many legitimate signatures—implies breaking various standard goals for the underlying hash function. Proving this implication should be within reach for formal verification, and violating these goals for the well-known SHA-256 hash function would be an astonishing breakthrough in cryptanalysis. Quantitatively, `sphincsf256sha256` is designed to provide 2^{128} post-quantum security even for a key that has signed 2^{64} messages (e.g., 1 million signatures per second for 584542 years).

`sphincsf256sha256` is only one of many signature systems provided by `libpqcrypto`. Other signature systems have features of interest in many applications. For example, the

[1] For example, 12 million devices around the Internet in 2014 were vulnerable to the "Misfortune Cookie" attack (http://mis.fortunecook.ie/), which exploited a bug that had been fixed by the implementor in 2005.

smallest signature size in `libpqcrypto`, just 45 bytes, is provided by `gui184`, a multivariate-quadratic signature scheme. The smallest total size of signatures and public keys, 3228 bytes, is provided by `dilithium2`, a lattice-based signature scheme. However, for OS updates, the `sphincsf256sha256` sizes are not a problem.

`libpqcrypto` makes each signature system available through a C interface, a Python interface, and a command-line interface; see D2.4 for full details. The command-line interface is fast enough for the OS-update application. The OS author generates a key pair as follows:

```
pq-keypair-sphincsf256sha256 5>update-publickey 9>update-secretkey
```

The OS author includes `update-publickey` (64 bytes) and `libpqcrypto` in the OS that you use. The OS author later signs an update file as follows, and distributes the signed file:

```
pq-sign-sphincsf256sha256 <file 8<update-secretkey >signedfile
```

After downloading an (allegedly) signed update file, your computer checks the signature as follows, using the `update-publickey` file that was already included in the OS:

```
pq-open-sphincsf256sha256 <signedfile 4<update-publickey >file
```

If an attacker has modified `signedfile` in transit then the `pq-open-sphincsf256sha256` command prints an error message and exits nonzero. Even if these signals are ignored, `pq-open-sphincsf256sha256` produces an empty output, so nothing bad will be installed.[2]

3.3 Combining pre-quantum and post-quantum systems

WP2's recommendation is, rather than simply replacing a pre-quantum signature system P with a post-quantum signature Q, to replace P with $P + Q$. Here $P + Q$ is the signature system defined as follows:

- The secret key for $P + Q$ is a pair of independent secret keys, one for P and one for Q. The public key for $P + Q$ is the corresponding pair of public keys.
- A $P + Q$ signature is a corresponding pair of signatures, one for P and one for Q.
- Verification for $P + Q$ means verification of *both* the P signature and the Q signature; i.e., forging a $P + Q$ signature requires forging both a P signature and a Q signature.

The $P + Q$ recommendation is equivalent to a restriction of the original recommendation: namely, replacing P with a post-quantum signature system Q', specifically the system $Q' = P + Q$. What is important about this restriction is that the auditor immediately sees that $P + Q$ is at least as secure as P.

For example, the auditor immediately sees that Ed25519+`sphincsf256sha256` is *at least* as secure as Ed25519. Anyone who forges an Ed25519+`sphincsf256sha256` signature has also successfully forged an Ed25519 signature, so an application that switches from Ed25519 to Ed25519+`sphincsf256sha256` has not lost security.

Another recommended alternative that fits better with a signed-message interface (see Section 3.2) is to use $P \circ Q$: a $P \circ Q$ signed message is the result of first using Q to sign the

[2] For comparison, a traditional verification interface reads two separate inputs, a message and an alleged signature, and then the user acts upon the same message, so if verification failures are ignored then the user will act upon forgeries.

original message, and then using P to sign the Q-signed message. There are many obvious variants involving signatures on hashes, signatures on randomized hashes, $Q \circ P$, etc., with slightly different performance properties but the same basic security property that an attacker is faced with the challenge of breaking both P and Q.

The extra size of the public key for $P + Q$ or $P \circ Q$ can draw complaints in contexts where it is important for public keys to be kept small. However, one can replace the public key for $P + Q$, or any other signature system, with a 256-bit hash of the public key, and include the original public key[3] inside each signature. The verifier checks that the original public key has the right hash, and then checks the rest of the signature.

3.4 The importance of auditing

Does deployment of Ed25519 $+ Q$ or Ed25519 $\circ Q$ or $Q \circ$ Ed25519 mean that we don't trust Q? On the contrary!

Our goal in designing a high-security post-quantum system Q is for that system *by itself* to be safe for the foreseeable future. Even in the pre-quantum world, hash-based signatures are confidence-inspiring *for the experts*, at least as confidence-inspiring as elliptic-curve signatures such as Ed25519. However, understanding this fact takes extra work *for the auditor*, while the auditor can immediately see that everything is still signed by Ed25519 and that the addition of Q has not lost security. This simplification of the auditing process has considerable value for deployment: it removes an unnecessary source of fear.

The long-term situation is different. If quantum computers scale as expected then users will see those computers breaking pre-quantum systems P more and more easily. In the meantime auditors will gain confidence in Q. The remaining value of including P will continue to decrease, and eventually will be outweighed by the simplicity benefit of switching from $P+Q$ to Q.

3.5 Alternatives

All of the signature systems in `libpqcrypto` follow the usual stateless signature API: the signer does not keep any notes regarding previously signed messages, not even the number of signatures. Compared to stateless hash-based signature systems such as `sphincsf256sha256`, *stateful* hash-based signature systems such as XMSS provide considerably better performance. However, `sphincsf256sha256` is already efficient enough for OS updates, as noted above, and has the advantage of being a drop-in replacement for existing signature systems. Software for `sphincsf256sha256` is shared between applications that can robustly maintain state and applications that cannot, while XMSS code is suitable only for applications of the first type.

The lack of performance pressure also means that hash-based signature systems could be combined with other post-quantum signature systems, but it is not clear whether the potential security benefits outweigh the cost in complexity. Note that adding $Q_1, Q_2, \ldots$ creates more software complexity than adding just one Q and reusing some P that is already deployed.

[3] The public key can be compressed slightly: for example, if one byte of the public key is omitted, the verifier can reconstruct that byte by trying all possibilities, only one of which will match the hash. Similar compression is also possible outside the context of a hash, but trying many signature-verification steps is usually more expensive than trying many hash-verification steps. There are other compression techniques, with or without hashes: for example, key generators can generate many keys until finding a key with a standard prefix, and then this prefix does not need to be transmitted.

4 A more challenging example: confidentiality, integrity, and availability for web browsing

The rest of this document emphasizes web browsing as an application, and considers the problems of protecting three types of security—confidentiality, integrity, and availability—against espionage, corruption, and sabotage:

- Confidentiality means that an attacker cannot figure out the contents of the transmitted data: for example, credit-card numbers remain secret. Beware that metadata (timing, length, etc.) is still exposed.
- Integrity means that incorrect data—any data forged or modified by an attacker—is detected, as in Section 3.
- Availability means that the correct data gets through. For example, an attacker who overloads a network with a flood of random packets is not compromising integrity but is denying service, compromising availability.

The presence of confidentiality in the list of security goals means that full deployment of a high-security post-quantum solution must take place *many years before* any quantum computers become available to attackers, where the number of years is the number of years that the user needs data to remain confidential.

These security goals for web browsing raise considerably more difficult performance challenges and integration challenges than the goal of protecting the integrity of software updates. This difficulty is reflected by how poorly the Internet is achieving these goals today against *pre-quantum* attackers, even without the additional challenges of post-quantum cryptography.

Consider, for example, the following type of attack: the attacker steals a server, and uses keys stored inside that server to retroactively decrypt stored ciphertext. HTTPS, the main security mechanism for web browsing on the Internet today, claims to provide "forward secrecy", quickly erasing keys so as to prevent those keys from being stolen. However, for various reasons explained by Langley in `https://www.imperialviolet.org/2013/06/27/botchingpfs.html`, HTTPS sessions often remain decryptable by HTTPS servers for many months.

As another example, there are no real availability protections in HTTPS. An attacker forging a single TCP "reset" packet will destroy an entire HTTPS connection, and an attacker forging a single DNS packet will prevent the connection from being made in the first place. The TLS software sees that something has gone wrong, but it has no way to recover. A browser using HTTPS can make a whole new connection, but this is slow and fragile. In short, a single forged packet causes huge damage. This attack is much less expensive, and optionally much more selective, than flooding a network.

The next section of this document reviews how Internet communication, especially web browsing, works *without* cryptography. The remaining sections analyze various options for cryptographic Internet protocols. In particular, the final section summarizes PQCRYPTO's "PQConnect" demo.

5 Non-cryptographic Internet communication

5.1 IP: Internet Protocol

Fundamentally, the Internet tries to communicate "packets"—limited-length byte strings—to specified "IP addresses".

For example, the web server `www.pqcrypto.eu.org` has IP address `131.155.70.18`. To contact this server, your computer creates an IP packet addressed to `131.155.70.18` and gives this packet to the Internet, which tries to deliver the packet to `131.155.70.18`.

It is important to understand that the Internet does not necessarily deliver an IP packet to its destination, even if the destination computer is alive and no attacks are in progress. The packet passes through a series of network links between intermediate computers. At any moment, these links and computers could be overloaded or malfunctioning, so an IP packet can be lost, or repeated, or randomly modified, or delivered later than a packet that was sent earlier.

The rest of this document uses the following model of allowable packet sizes: packets as large as 1280 bytes work reliably, while larger packets fail. This is not a perfect model of reality. What actually works is surprisingly complicated:

- The popular "Ethernet" standard for network links requires computers to handle 1500-byte packets. However, Internet packets often pass through network links that have somewhat smaller packet-size limits—for example, "tunnel" links that add extra data to each packet and then pass the resulting larger packets through Ethernet links.
- For IPv6 (version 6 of the Internet Protocol), 1280-byte packets are guaranteed to work by the standards and appear to work reliably, while 1400-byte packets are reported to cause frequent problems.
- For IPv4, 576-byte packets are guaranteed to work by the standards; 1280-byte packets appear to work reliably; 1492-byte packets are usually safe; 1500-byte packets are often safe.
- "WiFi" links allow somewhat larger packets. However, packets are randomly dropped for a variety of reasons; see, e.g., `http://ieeexplore.ieee.org/document/7317401/`.
- Some network links allow much larger packets. Computers typically probe the network (using "Path MTU discovery") to try to dynamically determine the largest packets that they can safely send at each moment, although this probing raises further reliability questions and security questions.

5.2 UDP: User Datagram Protocol

A UDP packet is an IP packet addressed to a particular "UDP port" inside a computer. Ports are indexed by 16-bit integers. Many ports are assigned to particular protocols: for example, port 53 is assigned to DNS (see Section 5.3), and port 67 is assigned to DHCP.

Internally, an OS allows a program to "bind" to a particular UDP port. The OS then delivers UDP packets to that program if the packets are addressed to that port. For example, a program providing DNS service binds to UDP port 53 and then receives packets addressed to UDP port 53. A program sending DNS requests instead temporarily binds to a random unused UDP port and receives responses addressed to that port.

5.3 DNS: Domain Name System

If you tell the browser on your laptop to connect to `www.pqcrypto.eu.org`, how does the browser learn the IP address `131.155.70.18`? The answer is that your laptop hears this address from a DNS server. Specifically, your laptop asks the `.pqcrypto.eu.org` name server, which has IP address `131.193.32.108`:

- Laptop → `131.193.32.108`: "`Where is www.pqcrypto.eu.org?`" This question is encoded as a DNS packet, which is a UDP packet sent to port 53. The packet also includes a return address (the laptop's IP address), along with a random UDP port of a program inside the laptop listening for the answer.
- `131.193.32.108` → laptop: "`131.155.70.18`"

Similarly, your laptop learned the IP address `131.193.32.108` of the `.pqcrypto.eu.org` name server by asking the `.eu.org` name server, which has address `46.226.109.38`:

- Laptop → `46.226.109.38`: "`Where is www.pqcrypto.eu.org?`"
- `46.226.109.38` → laptop (using the return address and port mentioned above): "`Ask the .pqcrypto.eu.org name server, 131.193.32.108`"

Before this, your laptop learned the address `46.226.109.38` by asking the `.org` name server, which has address `199.19.56.1`. Your laptop learned this address `199.19.56.1` by asking the Internet's root name server, which has the well-known IP address `198.41.0.4`.

For reliability, there are actually 13 IP addresses of root DNS servers, 6 IP addresses of `.org` DNS servers, etc. Your laptop sees a list of IP addresses, and randomly chooses one of the IP addresses to contact—perhaps disfavoring IP addresses that were recently unresponsive, and perhaps favoring IP addresses that have been fast to respond or that sound closer.

If the laptop does not receive a prompt response from a DNS server, it tries sending the same query again, typically to a different IP address. Perhaps the original query or response was lost because the network was overloaded, or because the original DNS server was down. Your laptop tries several times before it gives up.

5.4 TCP: Transmission Control Protocol

At first glance, TCP is just like UDP: a TCP packet is an IP packet addressed to a particular "TCP port" inside a computer. TCP ports, like UDP ports, are indexed by 16-bit integers. TCP port n is not attached to UDP port n, so TCP and UDP together provide 2^{17} ports.

However, TCP and UDP provide different interfaces to higher-level applications. A client program using TCP makes a TCP "connection" to a server and then sends a stream of bytes to the server. A server program "accepts" the connection and receives the *same* stream of bytes, even if packets are lost, repeated, or reordered. Meanwhile the server program sends a stream of bytes to the client, and the client program receives the same stream of bytes. Internally, TCP labels each byte with its position in the corresponding stream, acknowledges bytes that are received, and retransmits bytes that have not been acknowledged. For comparison, UDP (like IP) tries to deliver packets, but leaves it up to the application to handle acknowledgments and retransmissions, as illustrated by the DNS retransmissions described in Section 5.3.

Establishing a TCP connection involves three IP packets, exchanging two 32-bit random numbers:

- Client → server: “`SYN 168bb5d9`”
- Server → client: “`ACK 168bb5da, SYN 747bfa41`”
- Client → server: “`ACK 747bfa42`”

The server allocates memory for remembering the state of the connection. The client sends a stream of bytes, split into any number of packets, counting bytes from `168bb5da`. The server sends its own stream of bytes, split into any number of packets, counting bytes from `747bfa42`.

If the client does not receive a prompt SYNACK packet, it tries sending another SYN packet to the same IP address. After several tries, it gives up on that IP address and, if multiple IP addresses were provided, tries connecting to a different IP address. This strategy produces long delays when one of several servers is down. Clients could use a better strategy (as DNS clients do), trying all IP addresses before trying the first IP address again, but the design of TCP software interfaces makes this unnecessarily difficult.

If the client does not receive a prompt response later in the TCP connection, the client continues trying the same IP address,[4] and does not fall back to other IP addresses. This is important: other addresses are usually different machines that have no way to access the server's connection state. If the client keeps trying but still receives no response—perhaps the server has crashed—then it can give up on the connection and try making a connection to another address, but it has to start again with the TCP SYN packet.

5.5 HTTP: Hypertext Transfer Protocol

A user asks the browser on his laptop to display the web page `http://www.pqcrypto.eu.org/partners.html`. The laptop goes through several steps to retrieve this web page:

- It uses DNS (see Section 5.3) to find the address `131.155.70.18` of the web server. This involves several DNS packets.
- It makes a TCP connection (see Section 5.4) to port 80 of `131.155.70.18`. This involves three TCP packets.
- Inside that TCP connection, it sends an HTTP “GET” request, asking the server for the contents of `www.pqcrypto.eu.org/partners.html`. This is encoded as a 320-byte TCP data packet (which is retransmitted if it is not promptly acknowledged). The number of bytes depends on browser details, and the contents reveal information about the browser.
- Inside the same TCP connection, the server sends the web page. This involves two TCP packets, first 1500 bytes and then 1087 bytes. (The web page is actually 8095 bytes, but is compressed by the server to reduce network traffic.) There are also some 40-byte TCP ACK packets acknowledging receipt of data and closing the connection.

Each of these packets is subject to sabotage (being destroyed by an attacker), forgery (being replaced by a packet chosen by an attacker), and espionage (being observed by an attacker). For example, the attacker can modify the contents of the web page in transit.

[4] There are complicated rules to decide the exact retransmission schedule, avoiding network congestion.

5.6 Caching

Once the laptop has heard that www.pqcrypto.eu.org has IP address 131.155.70.18, it remembers this information for a while, so it can skip all the DNS packets for subsequent connections to www.pqcrypto.eu.org. This means that the laptop will not immediately see a change in the address. Administrators are generally willing to tolerate some delay in address changes. The amount of time for this information to be cached is chosen by the administrator of the .pqcrypto.eu.org DNS server.

Similar comments apply to higher-level DNS queries: for example, once the laptop has heard that .org has been delegated to 199.19.56.1, it remembers this information for a while, and avoids asking the root DNS servers for the same information again. The laptop also remembers the contents of the www.pqcrypto.eu.org web page, again for an amount of time chosen by the server administrator.

As part of the HTTP connection to www.pqcrypto.eu.org, the laptop can ask the HTTP server to keep the connection open, allowing the laptop to ask for another web page through the same connection. This avoids the overhead of creating and closing another connection, but extends the amount of time that the server is remembering state for this laptop, so it is not such a clear performance win.

Laptops are often configured to relay their DNS queries through caches ("DNS resolvers") operated by their Internet service providers, or through centralized Internet DNS caches such as Google's 8.8.8.8 or Cloudflare's new 1.1.1.1. Laptops are sometimes configured to similarly relay their HTTP queries through caches operated by their Internet service providers. This type of third-party caching relies on the third parties being able to see queries and replay the corresponding responses.

There is also caching on the server side. For example, www.fiat.com and www.ibm.com and www.rabobank.com are not operated by Fiat and IBM and Rabobank respectively: they are all operated by Akamai, a company that keeps copies of web pages on many computers that it operates around the world. Again this caching relies on Akamai being able to see queries and replay the corresponding responses.

5.7 More protocols

There are many Internet protocols beyond IP, UDP, DNS, TCP, and HTTP. For example, email is normally delivered through SMTP, the Simple Mail Transfer Protocol. An SMTP connection begins with DNS queries, like an HTTP connection,[5] and continues by making a TCP connection to port 25 of the server rather than port 80. Inside the TCP connection, the details of SMTP requests and responses are quite different from the details of HTTP requests and responses. Delivering a message through SMTP is conceptually similar to uploading a file to an HTTP server, but SMTP uses a simpler protocol focused on the task of mail delivery.

There is continual development of new protocols aiming at new features, such as better performance and better reliability. Protocols are typically differentiated by their choice of TCP port or UDP port: for example, the software company SAP registered TCP port 1128 for its "SAPHostControl" protocol. Reliability does not require TCP: for example, the Chrome browser often runs HTTP through Google's reliable QUIC protocol on UDP port 80.

[5] The DNS queries in SMTP ask specifically for a mail server. The mail server can have a different address from the web server. It can also have a list of addresses with different priorities.

6 How is cryptography triggered?

6.1 Opportunistic encryption: e.g., STARTTLS

Some protocols running over TCP have added options for "opportunistic encryption". The client tries sending a "STARTTLS" command; if the server supports this command, the client and server switch from TCP to TLS. This means that the client and server exchange keys and use those keys to encrypt the rest of the connection.

The most obvious problem with opportunistic encryption is that the attacker can simply disable it, modifying packets to remove the STARTTLS command. Internet service providers have already been observed doing this in the United States and elsewhere. See, e.g., https://privacyinternational.org/sites/default/files/2017-10/thailand_2017_0.pdf.

6.2 Trust on first use: e.g., HPKP

TOFU, "trust on first use", means that *if* the attacker does not bother to break the first connection then subsequent connections will be secure. For example, HPKP, "HTTP Public Key Pinning", is a mechanism for a server to send the server's public key to a client. The client then requires subsequent connections to the same server to be signed by that key (until a specified expiration time). Firefox supports HPKP, but Chrome is removing support.

TOFU might seem much better than opportunistic encryption if one imagines a long series of connections to a single site. However, TOFU means that each connection to a new site is a new opportunity for attack.

6.3 Server+client-side configuration per server+protocol: e.g., HTTPS

A user who replaces the URL http://www.pqcrypto.eu.org/partners.html with the URL https://www.pqcrypto.eu.org/partners.html is telling the browser to use HTTP-over-TLS-over-TCP rather than HTTP-over-TCP. The browser makes a TCP connection to port 443 instead of port 80. Within this TCP connection, the browser uses the TLS protocol to exchange cryptographic keys used to encrypt and authenticate subsequent data. Within this TLS connection, the browser runs HTTP.

The advantage of this manual configuration is that there is no opportunity for an attacker to disable the cryptography. The disadvantage is that manual work needs to be repeated for each protocol.

Consider, for example, a server administrator who enables https://pqcrypto.eu.org and then uses various secure channels to bring this change to the attention of everyone who had set up an http://pqcrypto.eu.org link. The server administrator then decides to enable TLS for another protocol, IMAP. This means taking the existing IMAP server running on TCP port 143, configuring it to also run over TLS on TCP port 993, and then telling users to change "`imap:`" in their IMAP-client configuration to "`imaps:`". The bigger picture is that a server supporting TLS for HTTP, IMAP, and a dozen other protocols needs clients to configure 14 protocol-specific security mechanisms.

There are many protocols where the client-side trigger to use TLS has not even been defined. For example, in principle DNS can run over TLS on TCP port 853, but how does a DNS client know that it is allowed to reach a DNS server this way? DNS is particularly important because the cryptographic protection for the https://pqcrypto.eu.org HTTP server, IMAP server, etc. does nothing to protect the DNS packets that told the client where

to *find* this server. An attacker can trivially deny service by forging these packets.[6] Some clients can be manually configured to contact particular DNS servers through TLS, but this does not help for servers that the client is contacting for the first time.

6.4 Server+client-side configuration per server: e.g., VPNs

The idea of a VPN, a "Virtual Private Network", is to let authorized users safely access an organization's private network as if the users were physically located within the network. The network administrator sets up a VPN server within the network, announces the IP address and public key of the VPN server, and configures a list of authorized users (typically identified by username and password) into the VPN server. Users run VPN clients that create "tunnels" to the server and send packets through those tunnels, decrypted by the server.

The concept of a private network is too restrictive to handle connections to many different public Internet servers such as `https://www.pqcrypto.eu.org`. But it is interesting to observe that VPN software does not need to be concerned with any of the differences between HTTP, DNS, etc. There are many protocols that have not been adapted to use TLS or any other cryptographic mechanism, but all of these protocols are protected by the VPN. The VPN works at a lower layer, simply encrypting and authenticating each packet.

Another advantage of packet-level encryption is that port numbers are hidden. Perhaps the attacker can figure out from packet sizes and timings which protocols are in use, but the attacker is not simply *given* this information. Having each packet authenticated also makes denial of service more difficult: a forged packet is discarded without doing damage.

6.5 Server-side configuration per server: e.g., MinimaLT

"DNSCurve" uses pre-quantum cryptography (the "X25519" ECDH system) to encrypt and authenticate DNS queries and responses. An administrator enables DNSCurve by *changing the name of the DNS server* to include a public key. This name is automatically retrieved by clients as part of normal DNS lookups, and triggers encryption by clients that support DNSCurve. The earlier DNS response that communicated the name of the server must also be protected, but this protection can also be handled by DNSCurve, avoiding TOFU weaknesses.

The same idea, obtaining public keys through server names, works straightforwardly for a wide range of protocols that refer to servers by name. It also works for large public keys if keys are replaced by hashes. Humans can still use friendly names such as `www.pqcrypto.eu.org`, as long as the DNS administrator links those names to public keys.

As in the case of VPNs (see Section 6.4), it is simpler and stronger to use one key per server than one key per server-protocol pair. The client simply encrypts each packet to the server, whether it is an HTTP packet, a DNS packet, or something else. The port number is hidden inside the packet. The server decrypts the packet and handles it appropriately. "MinimaLT" works this way, creating a single tunnel to handle all traffic between the client and the server.

[6]This is a problem with the lack of per-packet authentication. There is also, in some cases, a problem with the lack of encryption. The information revealed by DNS queries might be obvious from other packets, but it might not—consider, for example, a web server that hosts web sites under a huge number of different names. For many years browsers were revealing web-site names both through DNS and through "TLS SNI"; DNS developers were claiming that encrypting DNS was pointless since the same information was revealed by TLS SNI, and TLS developers were claiming that encrypting SNI was pointless since the same information was revealed by DNS.

7 Which protocol versions can the client and server choose?

7.1 Reasons for upgrading cryptographic choices inside protocols

It is important to distinguish three different types of cryptographic upgrades:

- Efficiency upgrades, not required for security.
- Controlled security upgrades (e.g., prohibiting substandard security levels) that might be needed someday but do not need to be rolled out in a rush.
- Emergency security upgrades, potentially requiring everything to be rebuilt in a rush.

Post-quantum cryptography is continuing to develop upgrades of the first type, and it will be important to roll out these upgrades as long as cost is limiting deployment. However, with careful choices it seems plausible that the risk of needing upgrades of the second type is low, and that the risk of needing upgrades of the third type is even lower.

There are three common arguments that cryptographic choices will never be able to stop changing:

- "Advances in computer power broke RSA-512, and then RSA-768. Whatever we deploy in the future will be broken by further advances in computer power."
- "Quantum computers will break much larger RSA keys. Whatever we deploy in the future will be broken by further advances in computer technology."
- "Advances in algorithms contributed to the RSA breaks. Whatever we deploy in the future will be broken by further advances in algorithms."

Each of these arguments has obvious flaws. There are standard analyses of the maximum amount of computation that can be carried out by the solar system, or even by the universe; it is not very expensive to build cryptographic systems where the best attacks known are beyond these security levels. The change from non-quantum computers to quantum computers matches the change from non-quantum physics (i.e., what can be simulated by non-quantum computers) to quantum physics (i.e., what can be simulated by quantum computers); unless quantum physics is terribly inaccurate, quantum computers are the end of the story. Finally, as deployed systems survive more and more cryptanalysis, it becomes more and more plausible that the systems are in fact secure against all possible attack algorithms.

This document makes the worst-case assumption that cryptographic protocols will in fact continue to change, at least for some time. However, this document also considers the desired long-term scenario of cryptographic stability, in which no further upgrades are necessary.

7.2 Single-version client, single-version server

It is simplest for the client to support only one version of a cryptographic protocol, and it is simplest for the server to support only one version of the cryptographic protocol. However, this combination works only if the client version matches the server version.

If upgrades are necessary, then all clients and servers need to upgrade at the same time. This is not easy to arrange. There will inevitably be outages on the designated "flag day".

7.3 Multiple-version client, single-version server

Allowing upgrades over a stretch of time requires *one* side, say the client, to support both the old and new versions. The server is then free to upgrade from the old version (and a public key for the old version) to the new version (and a public key for the new version). The following two rules ensure that this upgrade works:

- A clear moment is specified after which all clients are required to support the new protocol version. This does not mean clients should wait until this moment: on the contrary, clients can be upgraded months or years in advance if the protocol version is already specified (and should in any case upgrade far enough in advance to account for their local clock desynchronization). Similarly, servers do not need to rush to switch after that moment.
- All protocol versions define support for the protocol as including a responsibility for clients to support each new version of the protocol at the specified moment. Clients without fast enough upgrades must not deploy the protocol in the first place.

Furthermore, disabling the old version allows clients to be simplified, and is essential for a controlled security upgrade. Disabling the old version implies three further rules:

- A clear moment is specified after which servers are not permitted to use the old protocol version.
- All protocol versions define support for the protocol as including a responsibility for servers to stop using each version of the protocol after the specified moment. Servers without fast enough upgrades must not deploy the protocol in the first place.
- Another clear moment is specified by which clients are required to disable the old protocol version.

To summarize, for each protocol version there are three specified dates (D_1, D_2, D_3). The server is allowed to choose that protocol version only during dates $[D_1, D_2]$. The client starts supporting the protocol version at some point before D_1, and turns it off at some point during $[D_2, D_3]$. The following pattern allows a three-year client-upgrade schedule and a five-year server-upgrade schedule:

- Protocol version $(2028.07.01, 2033.07.01, 2036.07.01)$ is specified before 2025.07.01.
- Protocol version $(2029.07.01, 2034.07.01, 2037.07.01)$ is specified before 2026.07.01.
- Protocol version $(2030.07.01, 2035.07.01, 2038.07.01)$ is specified before 2027.07.01.
- Protocol version $(2031.07.01, 2036.07.01, 2039.07.01)$ is specified before 2028.07.01.
- Protocol version $(2032.07.01, 2037.07.01, 2040.07.01)$ is specified before 2029.07.01.
- Etc.

Version $(Y+1, Y+6, Y+9)$ can simply be defined to be identical to version $(Y, Y+5, Y+8)$ in the desirable case that the protocol has stabilized.

7.4 Single-version client, multiple-version server

Reversing the roles of the client and the server in the above description also produces a working system. Clients become simpler but servers become more complex; it is not clear whether this is a good tradeoff.

This approach has a disadvantage in the common case that the server announces a public key before hearing anything from the client: the public key needs to be compatible with all of the current protocol versions. One way to allow upgrades from public-key system X to public-key system Y is for the server to announce many public keys. This does not require extra bandwidth: the server can announce the public key $H(H(S_1), H(S_2), \dots)$ where $S_1, S_2, \dots$ are the server's actual public keys for X, Y, and perhaps more systems, and H is a hash function. The client then requests the key for the system it wants to use, and is given that key along with enough extra information to check the server's announced public key.

For comparison, in the situation of Section 7.3, a protocol upgrade can simply switch to a different type of public key. It might still be useful for the server's announced public key to be a hash so as to save space, but there is no need for the server to generate public keys for more than one system.

7.5 Multiple-version client, multiple-version server

Planning upgrades in advance, as in Sections 7.3 and 7.4, is not how TLS works.

New cryptographic choices in TLS are rolled out as options gradually added to some clients and gradually added to some servers. These options will be used if they are supported and preferred[7] by both the client and the server. Seeing a new option used immediately on a connection between an upgraded client and an upgraded server is pleasant and is referred to as "cryptographic agility". However, the big picture is much less pleasant.

As time passes, TLS servers develop more and more variation in which options they support. There is no general requirement for new options to be supported, and there is no general requirement for old options to disappear.[8] It becomes increasingly difficult for client software implementors to figure out which options are actually needed for interoperability; the implementors are pressured to support decades of options. Similarly, server software implementors are pressured to support decades of options.

The description of each cryptographic choice as a mere "option" makes it seem harmless to implement fast choices (e.g., the RC4 cipher) without worrying about security. The resulting security problems (e.g., attacks against RC4) are dismissed as merely being problems in one option, easily solved by switching to another option. However, even when there is consensus that an option is bad, safely turning off the option on the client (and server) side requires first upgrading all servers (and clients) to support some new option or set of options—and TLS does not have a process to do this. Some administrators begin turning off the old option anyway, creating interoperability problems.

[7] There is no global ordering of options: the client might prefer X to Y while the server prefers Y to X.

[8] TLS 1.3 has removed many ciphers, but there is no schedule to turn off TLS 1.2.

8 How does the server's public key authenticate the server?

8.1 Public-key encryption

In (e.g.) DNSCurve, the server's long-term public key is a key for an encryption system. The client encrypts fresh data to this public key, producing a session key shared by the client and server. The server's response is authenticated under this session key using secret-key techniques for message authentication (MACs). The attacker does not know the session key and cannot forge an authenticator for modified data.

`libpqcrypto` supports post-quantum public-key encryption, specifically various "key-encapsulation mechanisms" (KEMs). KEMs, by definition, produce ciphertexts that communicate session keys. All of the KEMs in `libpqcrypto` are designed to provide "IND-CCA2" security, meaning that the server's public key can be used for ciphertexts from any number of users. This means that a single KEM can be used for long-term authentication and for session encryption, simplifying the tasks of software engineering and auditing, although one might prefer to use different KEMs for performance reasons.

Similar "signature-free" authentication can also be used for the upgrade-integrity problem considered in Section 3, if the server can handle communication from each client. Instead of verifying a file's signature under the server's public key for a signature system, each client encrypts fresh data to the server's public key for an encryption system, and asks the server to authenticate the upgrade (or a hash of the upgrade) under the resulting session key.

Beware that there are also some KEMs that merely aim for "IND-CPA" security. For example, in the CECPQ1 experiment mentioned earlier, a New Hope public key was generated for each session and then thrown away. It would not have been safe to reuse the same public key for more users, since New Hope was not designed for IND-CCA2 security. ("New Hope CCA", included in `libpqcrypto`, *is* designed for IND-CCA2 security.)

8.2 Public-key signatures

In (e.g.) TLS, the server's long-term public key is a key for a signature system that signs the initial communication with the client. The client verifies the signature under this public key. The attacker does not know the server's corresponding secret key and cannot forge a signature on modified data. The signature also covers some fresh random data from the client, so the attacker cannot replay old signed data.

Using public-key signatures for authentication means that the client and server need to implement both a public-key signature system and (for confidentiality) a public-key encryption system. `libpqcrypto` supports various options for post-quantum signatures (see Section 3) and post-quantum encryption, but using two public-key primitives is more complicated than using just one.

For comparison, the CECPQ1 experiment mentioned earlier replaced the key exchange in TLS with a post-quantum key exchange, but it kept the pre-quantum signature scheme. Similarly, the prototype Open Quantum Safe library, `https://openquantumsafe.org`, integrates more key-exchange algorithms into TLS via the popular OpenSSL software library, but has not integrated post-quantum signatures into TLS.

9 How long are secret keys stored?

9.1 Forever

An attacker who steals a computer (a client or a server) can read all data stored on disk and, with some effort (see https://citp.princeton.edu/research/memory/), all data in RAM. Disk encryption does not help: the decryption key is in RAM. RAM encryption (such as Intel's Memory Encryption Engine) might help but is not available on most computers.

Data subsequently encrypted to that computer is decryptable by the attacker. Furthermore, *previous* ciphertexts sent to that computer, and intercepted by the attacker, are now decryptable by the attacker *if* the computer is still storing the secret key that it used to decrypt that data.

9.2 Until the end of a session

Standard practice is for a server's long-term public key to authenticate an "ephemeral" public key, and for user data to be encrypted solely to the ephemeral public key. If both sides erase all secrets (except the long-term secret key) at the end of the session, then they no longer have the ability to decrypt the ciphertexts. This erasure of keys provides some "forward secrecy": an attacker stealing a computer after the end of the session is unable to decrypt the session.

For example: The server's long-term public key is an encryption key S as in Section 8.1. The server sends a new public key E to the client.[9] The client sends new ciphertexts to both S and E to communicate two session keys k and k'. The client and server hash k and k' together to obtain a single session key k_1. The secret key corresponding to E is erased, along with k and k'. The session key k_1 is used to encrypt and authenticate subsequent packets, and is erased at the end of the session. An attacker that steals the server decrypts the ciphertext sent to the long-term key S, and thus computes k, but has no way to compute k' or k_1.

For efficiency, the server can reuse a public key E for many clients, for example generating a new key E every two minutes. This means that the server still has the power to decrypt the second ciphertext for two minutes after the start of the session, which could be up to two minutes after the end of the session, but users generally find this acceptable.

However, this reuse of E does not reduce the costs of transmitting E to many clients, having the clients encrypt to E, and decrypting the resulting ciphertexts. To reduce these session-initiation costs, sessions are stretched out for a much longer time, but this also means that the session key k_1 is not erased for a long time.

9.3 Until the next packet is received

Assume that the server authenticates an ephemeral public key E under a long-term public key S; that the client uses E to share a session key k_1 with the server; and that the server throws away the secret key corresponding to E. At this point the client and server can now use k_1 to encrypt packets. These packets are decryptable by either side's copy of k_1, and not by any other keys.

[9] An alternative is for the client to send an ephemeral public key to the server, and for the server to send a ciphertext to this key. This has the advantage of balancing cryptographic work and bandwidth between the client and server. It has the disadvantage that both sides need software for all three public-key operations: key generation, encryption, and decryption. For comparison, if the server sends both S and E, then the server does not need encryption software, and the client does not need key-generation or decryption software.

If the client and server both overwrite k_1 with a new key k_2, then they lose the ability to decrypt packets encrypted to k_1. This key k_2 can be generated randomly by the client and sent to the server, or generated randomly by the server and sent to the client, or computed as a one-way hash of k_1. Repeating this process prevents decryption of packets encrypted to k_2, then prevents decryption of packets encrypted to the next key k_3, etc.

To limit the coordination required between the two sides, it is helpful to have one chain of keys $c_1, c_2, \ldots$ used for client encryption, and a separate chain of keys $s_1, s_2, \ldots$ used for server encryption. The client encrypts its nth packet under key c_n, and can immediately erase that key.[10] The server erases key c_n once it has successfully received and decrypted this packet. Similarly, the server encrypts its nth packet under key s_n.

Replacing a key c_1 with a hash, $c_2 = H(c_1)$, is an example of "hash ratcheting", as used in the "Silent Circle Instant Messaging Protocol". A tweak used in the "Axolotl" protocol, also known as the "Double Ratchet" protocol,[11] is that the nth packet is encrypted under key $c'_n = H'(c_n)$, so that a server receiving the nth packet without having received the $(n-1)$st packet can keep key c'_{n-1} to decrypt the $(n-1)$st packet and key c_{n+1} to decrypt subsequent packets, while erasing all of the keys c_{n-1}, c_n, c'_n that can be used to decrypt the nth packet.

One can replace the linear chain of hashes $c_1, c_2, \ldots$ with a structure that allows some "fast forwarding": for example, obtaining c_{16k+1} as a (different) hash of c_{16k-15} rather than as a hash of c_{16k}. This saves time if many packets are lost, and reduces the worst-case CPU cost of rejecting a forged packet with large n.

9.4 Until two minutes later

An attacker that floods a network prevents the server from erasing any keys in Section 9.3. Meanwhile the client continues sending more and more user data. The attacker records the ciphertexts, steals the server, and decrypts the ciphertexts the same way that the server would have.

Packets sent *after* the server is stolen would have been decryptable in any case. By flooding the network, the attacker extends the decryptability back in time to the beginning of the flood.

One way to limit the extent of this exposure is for each computer to put a time limit on each key, for example erasing the key at most two minutes after the first moment that user data could have been encrypted using that key. This time limit is a user-comprehensible security property: two minutes after a ciphertext has been generated, nobody can decrypt the ciphertext. Two minutes are measured on the computer's local clock, which might be slightly fast or slightly slow but can be protected from outside corruption.

For example, the client can use key c_1 until time 30 (measured in seconds on the client's clock from the start of the session), then key c_2 until time 60, then key c_3 until time 90, etc. The server throws key c_1 away at time 120 (measured in seconds on the server's clock from the start of the session), throws key c_2 away at time 150, throws key c_3 away at time 180, etc. The 90-second gap allows for some network delays and some clock skew.

[10] The client keeps a copy of the encrypted packet to retransmit until the server acknowledges the packet. This does not require the client to keep the key.

[11] "Double" does not refer to the two hashes per message. Instead it refers to the combination of hash ratcheting with "public-key ratcheting", which means using new public-key operations to share new randomness in case previous randomness is somehow compromised in a way that does not compromise future randomness. This type of compromise—also considered in "backward security" and "key-compromise impersonation" and "post-compromise security"—is not the same as the computer-theft scenario mentioned earlier.

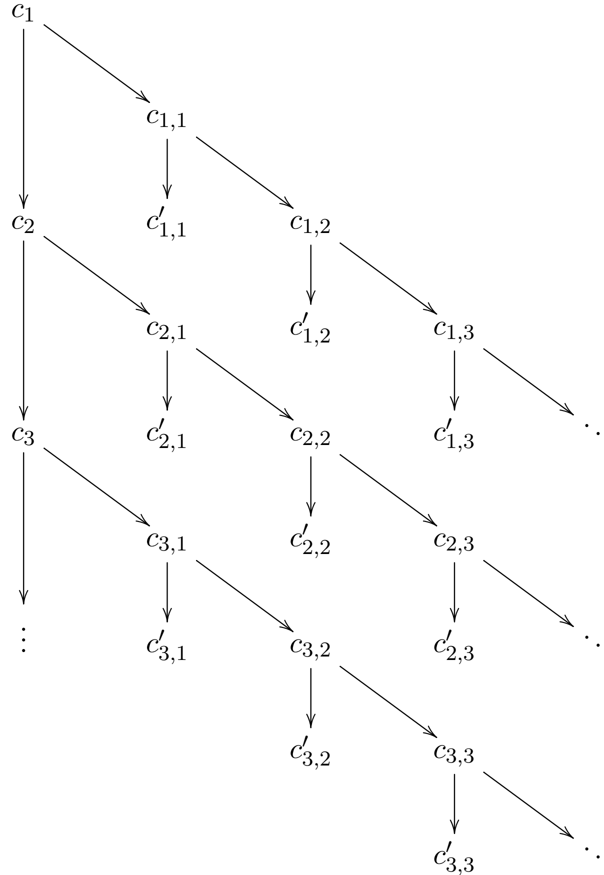

Figure 9.1: Erasing keys as soon as they are used, and in any case within two minutes. Key $c'_{1,i}$ is used for the ith client packet between time 0 and time 30, and is erased by the server as soon as it is used, or at the latest at time 120. Key $c'_{2,i}$ is used for the ith client packet between time 30 and time 60, and is erased by the server as soon as it is used, or at the latest at time 150.

If the client's clock is faster than the server's clock then the server may receive data encrypted under key c_n before the server's clock has reached time $30n$. If the server receives data encrypted under the next key c_{n+1} before the server's clock has reached time $30n$, then it knows that the client's clock is at least 30 seconds ahead of the server's clock, and it speeds up to compensate: for example, it subtracts 10 seconds from its forthcoming alarms saying when to move to new keys (for sending and receiving). If the server similarly sees that the client's clock is falling *behind* the server's clock, then it does not *slow down* movement to new keys; instead it makes sure to ping the client so that the client (following analogous rules for data sent in the other direction) will speed up.

The next-packet rule in Section 9.3 can be combined with this section's two-minute rule as follows. There are again keys $c_1, c_2, c_3, \ldots$, each used for 30 seconds. Key c_n is used to create a chain of keys $c_{n,1}, c_{n,2}, \ldots$, or more generally a tree as discussed above. The client uses key $c_{n,i}$ (or a hash of this key, as in Axolotl) for the ith packet in the nth 30-second interval. The server discards all keys $c_{n,i}$ at time $30n + 90$. See Figure 9.1.

10 How does the client resume a session?

10.1 No resumption

A client has previously downloaded a web page from a web server. There has been no traffic between the client and the server for some time. The server has discarded information about the original connection. Perhaps the client has a new IP address.

The client now wants to download another web page from the same server. The simplest way to handle this is for the client to start anew, using public-key encryption to set up a new session key, as if the client had never contacted the server before.

Setting up a new key has the benefit of not linking the new page request to the old page request. On the other hand, often the client is known to the server anyway, or the requests are linked for other reasons. If "forward secrecy" is desired—see Section 9.2—then setting up a new key incurs the cost of obtaining the server's latest ephemeral key.

10.2 Refreshing the session key

Assume that the server is willing to store a symmetric key r, a "resumption key", for the client. The server and client agreed on this key via the original encrypted connection, along with a session identifier that the server can use to locate the stored key. The client later resumes the session by sending this identifier back to the server and authenticating data under r, without any public-key operations. TLS "session IDs" follow this approach. Generating a new random session identifier at each resumption prevents third parties from using the client identifier to link several resumptions.

If the client and server have not been constantly ratcheting r then encrypting user data under r would violate the two-minute rule of Section 9.4. The client can instead send a fresh random session key to the server, encrypted and authenticated under r. The server overwrites r with $H(r)$ and sends a confirmation back to the client. Upon receiving the confirmation, the client replaces its own copy of r with $H(r)$ and begins sending user data under the new session key. Now neither side can decrypt the ciphertext that communicated the session key; and the session key has the time limits explained in Section 9.4, so it will be erased soon.

A subtlety here is that the confirmation might be lost. The client can again try sending a fresh random session key under r, but now the server has erased r. The client can try both r and $H(r)$, and then $H(H(r))$, etc., but this scales poorly when there is a network outage. A solution is for the server to record the client's packet and server's confirmation, and for the client to replay exactly this packet until obtaining the confirmation. This replaying can stretch over any number of ratcheting periods; the packet includes the time when the packet was created (measured from, e.g., the beginning of the session), allowing the client and server to ratchet r appropriately.

10.3 Eliminating server storage

Instead of storing data locally, the server can store the data as a "cookie" on the client. A cookie is an authenticated ciphertext under a symmetric key, the "cookie key", known only to the server. In other words, the server is sending encrypted data to a future version of the server via the client.

In particular, the server can send the client's resumption key r as a cookie to the client. Later, to resume the session, the client sends this cookie back to the server (without further

encryption) and begins encrypting and authenticating data under this key. TLS "session tickets" follow this approach.

If "forward secrecy" is desired then it is important for the server to promptly erase the cookie key. Otherwise an attacker subsequently stealing the server obtains the cookie key and can decrypt the cookie, obtaining the new session key. However, erasing the cookie key prevents the server from decrypting other cookies using this key. The server can maintain a large pool of cookie keys, but the approach in Section 10.2 seems more efficient.

11 PQConnect

A user types "`www.pqcrypto.eu.org`" into his laptop's web browser. In the PQConnect demo, the laptop automatically uses high-speed high-security post-quantum cryptography to connect to the `www.pqcrypto.eu.org` web server. The browser displays the same page that it would have displayed without PQConnect. Browsing to non-PQConnect web sites also works normally, without post-quantum cryptography.

This demo does not use any changes to the browser software. The user makes a one-time configuration change to the browser before the demo, namely telling the browser to use TCP port 1080 of IP address `127.0.0.1` as a local "SOCKSv5 proxy". This means that the browser routes all outgoing connections through this proxy. The proxy is a program `pqconnect-client` (written by PQCRYPTO) running on the user's computer; IP address `127.0.0.1` is a special address meaning "this computer".

`pqconnect-client` automatically detects that the `www.pqcrypto.eu.org` server supports PQCRYPTO's PQConnect protocol. The user does not tell `pqconnect-client` about this. Instead the server administrator enables PQConnect before the demo by

- installing another program `pqconnect-server` (written by PQCRYPTO);
- telling the program to accept PQConnect for `www.pqcrypto.eu.org`; and
- modifying the `www.pqcrypto.eu.org` information published through DNS.

This modified `www.pqcrypto.eu.org` information in DNS is how `pqconnect-client` detects that `www.pqcrypto.eu.org` supports PQConnect. Note that for security it is important for these DNS lookups to also be secured, for example through preliminary PQConnect connections to the DNS servers; this is not part of the demo.

If a server administrator sets up PQConnect for another site, then the same browser will, via `pqconnect-client`, automatically use PQConnect for that site too.

Internally, PQConnect follows the model from Section 7.3: multiple-version client and single-version server. Servers are permitted to use the current protocol version, PQConnect 2018.01, until 2018.07.01, and clients are required to disable this version at some point between 2018.07.01 and 2018.09.01. The following comments are specific to PQConnect 2018.01.

The PQConnect server has two long-term public keys: a 1357824-byte post-quantum `mceliece8192128` public key and a 32-byte pre-quantum X25519 public key. The McEliece key is intended as the foundation of security: it is used to encrypt everything, and also to authenticate the server. The X25519 key is used as another layer to encrypt everything and to authenticate the server; this is intended to let auditors easily see that PQConnect is as secure as X25519, assuming that the symmetric components are also secure. To simplify this type of combination, `libpqcrypto` supports X25519 under the name `x25519notpq`.

A 32-byte hash of the two public keys is encoded as an ASCII string beginning with the magic symbols `pq1` and ending with `.pqcrypto.eu.org`. The server administrator adds this new name into DNS with the server address `131.155.70.18`, and changes the original name `www.pqcrypto.eu.org` to a "CNAME" pointing to the new name.[12] The structure of DNS provides this CNAME to the client without any extra requests from the client. `pqconnect-client`, when asked by the browser to make a connection, sees the encoded hash in the information provided by DNS, and switches to the PQConnect protocol.

`pqconnect-client` now asks the server for the two public keys. This request to the server is split into a series of packets, each requesting a small component of the public keys. The hash function is designed as a limited-width tree so that each component can be verified immediately:

- The 1357856-byte McEliece key is split into 1179 parts, each part (before the last) having 1152 bytes.
- Each part is hashed separately, producing 37728 bytes.
- These 37728 bytes are split into 33 parts, again each part (before the last) having 1152 bytes.
- Each part is hashed separately, producing 1056 bytes.
- These 1056 bytes and the 32-byte X25519 key are hashed, producing the final 32-byte hash.

The client requests the 1056 bytes and 32-byte X25519 key; then requests each of the 33 parts; then requests each of the 1179 parts. Any response packet that does not match its expected hash is discarded. The client optionally saves the results in a cache of public keys (indexed by the hash) so that it can skip these requests for future connections to the same server.[13]

`pqconnect-client` next generates a KEM ciphertext to each of the server's long-term public keys: a 240-byte McEliece ciphertext and a 32-byte X25519 ciphertext. It sends these two ciphertexts together in a single packet requesting an ephemeral public key from the server. Subsequent data exchanged between the client and server is encrypted and authenticated under a master session key obtained by hashing the two session keys communicated by these two ciphertexts.

For efficiency, the server switches to a different public-key system for its ephemeral keys: `sntrup4591761` (1218-byte public key) combined again with X25519 (32-byte public key). These two public keys are encrypted and authenticated to form the server's response packet.

The client now generates a 1047-byte KEM ciphertext to the `sntrup4591761` ephemeral key, and a 32-byte ciphertext for X25519. Both sides hash the previous master key together with the two session keys communicated by these two ciphertexts to form a new master session key. The server erases the ephemeral secret keys, so it cannot decrypt these ciphertexts later.

The protocol does not support resumption. When the server's long-term public key is cached, simply starting a new session involves one packet to the server, one packet back, and then a followup packet to the server. This has essentially the same efficiency as starting a TCP connection.

[12]The design of DNS does not allow CNAMEs for names at the top of delegated trees. For such names, PQConnect instead puts a hash into the name of the DNS server, as in DNSCurve.

[13]The same public keys could alternatively be obtained from other servers or from broadcast mechanisms.