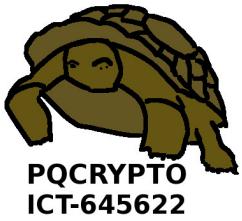

PQCRYPTO

Post-Quantum Cryptography for Long-Term Security

Project number: Horizon 2020 ICT-645622

D2.2

Internet: Preliminary integration

Due date of deliverable: 1. March 2017
Actual submission date: 3. 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 1.0

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: Preliminary integration

Daniel J. Bernstein

3. April 2018
Revision 1.0

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 is a progress report on WP2's Task 2.3, developing new Internet protocols and modifying existing protocols to add post-quantum security. This document will be superseded by D2.5.

Keywords: protocols, Internet, post-quantum cryptography

Contents

1 Examples of typical Internet applications

1.1 Software updates

Your computer downloads a new version of its operating system. Your computer checks a signature on the download from the OS manufacturer: otherwise attackers could insert malware into the OS. This is a critical use of cryptography. For example, OpenBSD updates are signed using Ed25519, a state-of-the-art ECC signature system.

This pre-quantum signature system P needs to be replaced with a post-quantum signature system Q. Auditors are happier if P is instead replaced with $P + Q$:

- The $P + Q$ public key concatenates the P public key and the Q public key.
- The $P + Q$ signature concatenates the P signature and the Q signature.

Sometimes it is important for public keys to be kept small. One can replace the public key with a 256-bit hash, and include the original public key (possibly compressed) inside the signature. The verifier checks that the original public key has the right hash, and then checks the rest of the signature.

Consider, for example, Ed25519+SPHINCS-256:

- A SPHINCS-256 hash-based signature is 41KB; ≈50 million cycles to generate on a typical CPU; ≈1 million cycles to verify. This is negligible cost to sign, transmit, and verify compared to the OS update.
- Adding Ed25519 has unnoticeable cost in CPU time and network traffic. It adds some extra system complexity, but the system includes Ed25519 code anyway.

The auditor sees very easily that the security of Ed25519+SPHINCS-256 is *at least* the security of Ed25519. Anyone who can forge an Ed25519+SPHINCS-256 signature can also forge an Ed25519 signature.

Does deployment of $P + Q$ mean that we don't trust Q? On the contrary:

- The pre-quantum situation is that hash-based signatures are even more confidence-inspiring than ECC signatures. But understanding this fact takes extra work for the auditor, whereas it is easy to see that everything is still signed by P.
- The long-term situation is that users see quantum computers easily breaking P. We then simplify the system by switching from $P + Q$ to Q.

1.2 The World Wide Web

Your browser connects to a web server; downloads web pages; sometimes uploads data. Your basic security goals in this scenario are integrity, availability, and confidentiality.

For example, users searching for health information generally do not consider it acceptable if an Internet service provider—or anyone else controlling a machine on the path between the browser and the web server—sells the resulting health-search data to insurance companies and advertisers; or blocks the resulting web pages; or modifies the resulting web pages.[1]

[1] An advanced security goal is for even *the web site itself* to not know which user is obtaining data. This document focuses on the simpler problem of protecting content against third parties; there are higher-level tools that build on this foundation to also protect metadata.

The main security mechanism used today for web browsing is HTTPS, i.e., HTTP over TLS. TLS relies on two main tools from public-key cryptography, namely encryption and signatures. It might sound easy to add post-quantum encryption systems and post-quantum signature systems: for example, replacing RSA encryption with RSA+McEliece and replacing RSA signatures with RSA+SPHINCS-256.

However, TLS has a long history of security being damaged by the pursuit of performance. The performance of ECC—small network traffic and small CPU time for high *pre-quantum* security—has helped considerably, but the security of TLS is still limited by structures built for slower cryptography: this is what has led to, e.g., the structural inability of a TLS connection to survive a single forged packet. Furthermore, the deployment of *post-quantum* cryptography in TLS is still limited by performance concerns.

The rest of this document looks at what Internet cryptography actually needs. How does the Internet communicate data? What are the fundamental costs of protecting this data against espionage, corruption, and sabotage in a post-quantum world?

2 How the Internet communicates data today

2.1 IP: Internet Protocol

IP communicates "packets": limited-length byte strings.

Each computer on the Internet has a 4-byte "IP address"; e.g., `www.pqcrypto.eu.org` has address `131.155.70.18`.

Your browser creates a packet addressed to `131.155.70.18`, and gives this packet to the Internet. Hopefully the Internet delivers that packet to `131.155.70.18`.

2.2 DNS: Domain Name System

You actually told your browser to connect to `www.pqcrypto.eu.org`. How did the browser learn the IP address `131.155.70.18`?

Answer: your browser asked a "domain name server". Specifically, your browser asked the `.pqcrypto.eu.org` name server, which has IP address `131.193.32.108`:

- Browser → `131.193.32.108`: "`Where is www.pqcrypto.eu.org?`"
- The IP packet from your browser also includes a return address: the IP address of your computer.
- `131.193.32.108` → browser: "`131.155.70.18`"

How did your browser learn the name-server address, `131.193.32.108`? Answer: your browser asked the `.eu.org` name server, which has IP address `46.226.109.38`:

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

How did your browser learn the `.eu.org` name-server address, `46.226.109.38`? Answer: your browser asked the `.org` name server, which has IP address `199.19.54.1`:

- Browser → `199.19.54.1`: "`Where is www.pqcrypto.eu.org?`"
- `199.19.54.1` → browser: "`Ask the .eu.org name server, 46.226.109.38`"

Similarly, your browser learned "`199.19.54.1`", the `.org` server address, by asking the "root" name server. The root name-server address is widely known.

2.3 TCP: Transmission Control Protocol

Packets are limited to 1280 bytes. (Actually, the limit depends on the network. For IPv4, nothing above 576 is guaranteed to work by the standards, but 1280 works reliably; usually 1492 is safe, often 1500, occasionally more. For IPv6, 1280 is guaranteed to work by the standards, and there are reports of problems with 1400.)

The page you're downloading from `www.pqcrypto.eu.org` doesn't fit in a packet. The browser actually makes a "TCP connection" to `www.pqcrypto.eu.org`, starting with an exchange of random numbers:

- Browser → server: "`SYN 168bb5d9`"
- Server → browser: "`ACK 168bb5da, SYN 747bfa41`"
- Browser → server: "`ACK 747bfa42`"

The server now allocates memory for this TCP connection. The browser sends a stream of data, split into any number of packets, counting bytes from `168bb5da`. The server sends its own stream of data, split into any number of packets, counting bytes from `747bfa42`.

The data sent by the browser inside this TCP connection is an HTTP request. The data sent by the server is a response to the request.

The main feature advertised by TCP is "reliable data streams". A TCP connection isn't confused when the Internet loses packets or delivers packets out of order: the computer checks the counter inside each TCP packet. The computer retransmits data if the data is not acknowledged. There are complicated rules to decide the retransmission schedule, avoiding network congestion.

3 Pre-quantum Internet cryptography

3.1 Stream-level cryptography and packet-level cryptography

`http://www.pqcrypto.eu.org` uses HTTP over TCP.

`https://www.pqcrypto.eu.org` instead uses HTTP over TLS over TCP. Your browser uses DNS (see above) to find the IP address `131.155.70.18`; makes a TCP connection; inside the TCP connection, builds a TLS connection by exchanging cryptographic keys; inside the TLS connection, sends an HTTP request and receives a response.

What happens if an attacker forges a DNS packet pointing to a fake server? Or a TCP packet with fake data, such as a "reset" terminating the connection?

The DNS software is fooled. The TCP software is fooled. The TLS software sees that something has gone wrong, but has no way to recover. The browser using TLS can make a whole new connection, but this is slow and fragile. To summarize, there was huge damage from a single forged packet.

The modern trend (see, e.g., DNSCurve, CurveCP, MinimaLT, and Google's QUIC) is to authenticate and encrypt each packet separately. A forged packet is discarded immediately, without doing any damage. A packet is retransmitted if there is no *authenticated* acknowledgment. Packet-level cryptography also has an engineering advantage of working for more protocols than stream-level cryptography. However, it raises the important question of what to do if authentication and encryption do not fit into a packet.

3.2 The KEM+AE philosophy

The original view of RSA is that a message m is encrypted as $m^e \bmod pq$. This is vulnerable to a wide range of attacks in the usual case that m is partially known; it also requires messages to be shorter than public keys.

A more modern "hybrid" view of RSA, including random padding, is as follows:

- Choose a random AES-GCM key k.
- Randomly pad k as r.
- Encrypt r as $r^e \bmod pq$.
- Encrypt m (which now can have any length) under k.

This approach is fragile and has many problems depending on details: consider, e.g., the Coppersmith attack, the Bleichenbacher attack, and the incorrect OAEP security proof.

Shoup's "KEM+DEM" view uses RSA differently:

- "Key encapsulation mechanism": Choose a random $r \bmod pq$. Encrypt r as $r^e \bmod pq$. Define $k = H(r, r^e \bmod pq)$ where H is a hash function.
- "Data encapsulation mechanism": Encrypt and authenticate m under this AES-GCM key k. The authenticator catches any modification of $r^e \bmod pq$.

This is much easier to get right than the earlier "hybrid" approach. It is also easier to adapt: for example, if P and Q are KEMs, one can build $P + Q$ by simply hashing together the session keys produced by P and Q.

The standard DEM security hypothesis is a weak single-message version of security for secret-key authenticated encryption. This does not guarantee that it is safe to reuse k for multiple messages. However, if DEM security is replaced by full AE security, then there is no problem: i.e., KEM+AE allows the session key k to be reused for multiple messages.[2] AES-GCM, Salsa20-Poly1305, etc. aim for full AE security.

A more complicated alternative is to use a KEM+DEM to encrypt another secret key k', and then use k' as an AE key.

3.3 DNSCurve: ECDH for DNS

The existing DNSCurve protocol encrypts DNS queries and responses as follows:

- The server knows an ECDH secret key s.
- The client knows an ECDH secret key c and the server's public key $S = sG$.

[2] This statement should be within reach of current formal-verification techniques.

- Client → server: packet containing cG and $E_k(0, q)$. Here q is a DNS query; E is an authenticated cipher; $k = H(cS)$; and H is a hash function.
- Server → client: packet containing $E_k(1, r)$ where r is a DNS response.

The client can reuse c across multiple queries, but this leaks metadata. Let's assume c is used only once. Then the same communication can be viewed in the KEM+AE framework as follows:

- The client is sending $k = H(cS)$ encapsulated as cG. This is an "ECDH KEM".
- The client then uses k to authenticate and encrypt a DNS query q.
- The server also uses k to authenticate and encrypt a DNS response r.

4 Post-quantum Internet cryptography

4.1 Post-quantum encrypted DNS

Ongoing PQCRYPTO research includes various choices of KEMs: for example, some KEMs based on McEliece, and some KEMs based on NTRU. One can try to drop any of these KEMs into the protocol in Section 3.3 as follows:

- Client → server: packet containing C and $E_k(0, q)$. Here C is a ciphertext for the KEM encrypted to the server's public key, and k is the session key communicated by this ciphertext.
- Server → client: packet containing $E_k(1, r)$.

The basic security properties of this protocol are as follows:

- Confidentiality: The attacker cannot compute k, and cannot decrypt $E_k(0, q), E_k(1, r)$.
- Integrity: The server never signs anything, but E_k includes authentication. The attacker can send new queries but cannot forge q or r. The attacker *can* replay a query, presumably producing the same encrypted response.
- Availability: If the attacker forges a packet to the client, the client discards the forgery and continues waiting for the legitimate reply. Eventually the client retransmits the query.

One difficulty is that the DNS response r normally states an address of a server and the public key of that server. What happens if this key is too long to fit into a single packet? A simple answer is that the client separately requests each block of the public key. The client can do many of these requests in parallel.

Another difficulty is that (depending on lengths) the secret-key ciphertext $E_k(0, q)$ might not fit into the same packet as the public-key ciphertext C. A way out of this difficulty is to use "cookies":

- The client sends C and a short $E_k(0, q')$ containing a **cookie request** q'.
- The server sends $E_k(1, r')$ containing a **cookie** r': the server state (including k) encrypted from the server to itself. The server can now forget this state.

- The client sends a packet $r', E_k(2, q)$. The server recovers the state and decrypts.
- The server sends $E_k(3, r)$.

This puts q into a separate packet from C, while q' and r' are relatively short.

4.2 Generalizations and adaptations

The same strategy works for protecting connections. The two-packet $C \to S$, $S \to C$ data flow used above is not special; one can reuse k for many packets each direction.

Another TCP availability problem is that the server allocates memory for each connection; if there are too many connections then the server runs out of memory. Here are several possible responses:

- Semi-solution ("SYN cookies"): Allocate memory only after the client sends r'. This protects only against blind attackers.
- Solution 1: The client sends hashcash to pay for the server's memory.
- Solution 2: Redo protocols to avoid state on the server. A good model here is NFS rather than HTTP.
- Solution 3 for, e.g., SSH: Allocate memory only for authenticated clients.

The server can authenticate the client without signatures, the same way that the client authenticates the server. The server sends an encapsulation of a new key k' to the client's public key, hashes k' into a shared session key, and checks subsequent authenticators from the client under the new session key.

4.3 Big keys

The McEliece public key is 1MB for long-term confidence today. Encryption mechanisms with smaller keys have been less thoroughly studied. Is 1MB a performance problem?

For comparison, the size of an average web page in the Alexa Top 1000000 is 1.8MB. A web page often needs public keys for several servers, but the public key for a server can be reused for many pages.

The most important limitation on reuse of public keys is that one wants to switch to new keys and **promptly erase old keys**. The rationale for this key erasure is "forward secrecy": subsequent theft of the computer doesn't allow decryption of ciphertexts encrypted to the earlier keys.

For example, Microsoft SChannel switches keys every two hours. Here is the performance of a new key every hour:

- If the server makes the new key: the server pays for key generation once per hour (or less frequently if the server is idle); the client encrypts to the new key; the server decrypts.
- If the client makes the new key: the client has the key-generation cost; the server has the encryption cost; the client has the decryption cost.

Either way, there is one key transmission per hour for each active client-server pair. This is 2222 bits per second if keys are 1MB; i.e., 22Mbps for a server with 10000 active clients.

How does a *stateless* server encrypt to a new client key without storing the key? One answer is to slice the McEliece public key so that each slice of encryption produces a separate small output. The client sends slices (in parallel), receives outputs as cookies, and sends cookies (in parallel). The server combines these cookies. This process continues up through a tree of (parallel) combinations. To guarantee integrity of the computation, the server generates randomness as a secret function of a hash of the key, and statelessly verifies this hash.