The post-quantum Internet

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

Includes joint work with:

Tanja Lange

Technische Universiteit Eindhoven

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.org` has address `131.155.70.11`.

Your browser creates a packet addressed to `131.155.70.11`; gives packet to the Internet. Hopefully the Internet delivers that packet to `131.155.70.11`.

DNS: Domain Name System

You actually told your browser to connect to `www.pqcrypto.org`.

Browser learns "`131.155.70.11`" by asking a name server, the `pqcrypto.org` name server.

Browser → `131.155.71.143`:
"`Where is www.pqcrypto.org?`"

DNS: Domain Name System

You actually told your browser to connect to `www.pqcrypto.org`.

Browser learns "`131.155.70.11`" by asking a name server, the `pqcrypto.org` name server.

Browser → `131.155.71.143`:
"`Where is www.pqcrypto.org?`"

IP packet from browser also includes a return address: the address of your computer.

`131.155.71.143` → browser:
"`131.155.70.11`"

Browser learns the name-server address, “`131.155.71.143`”, by asking the `.org` name server.

Browser → `199.19.54.1`:
“`Where is www.pqcrypto.org?`”

`199.19.54.1` → browser:
“`Ask the pqcrypto.org name server, 131.155.71.143`”

Browser learns the name-server address, "`131.155.71.143`", by asking the `.org` name server.

Browser → `199.19.54.1`:
"`Where is www.pqcrypto.org?`"

`199.19.54.1` → browser:
"`Ask the pqcrypto.org`
`name server, 131.155.71.143`"

Browser learns "`199.19.54.1`", the `.org` server address, by asking the root name server.

Browser learns the name-server address, "`131.155.71.143`", by asking the `.org` name server.

Browser → `199.19.54.1`:
"`Where is www.pqcrypto.org?`"

`199.19.54.1` → browser:
"`Ask the pqcrypto.org name server, 131.155.71.143`"

Browser learns "`199.19.54.1`", the `.org` server address, by asking the root name server.

Browser learned root address by consulting the Bible.

TCP: Transmission Control Protocol

Packets are limited to 1280 bytes.

(Actually depends on network.
Oldest IP standards required
$\geq$576. Usually 1492 is safe,
often 1500, sometimes more.)

TCP: Transmission Control Protocol

Packets are limited to 1280 bytes.

(Actually depends on network. Oldest IP standards required $\geq$576. Usually 1492 is safe, often 1500, sometimes more.)

The page you're downloading from `pqcrypto.org` doesn't fit.

TCP: Transmission Control Protocol

Packets are limited to 1280 bytes.

(Actually depends on network. Oldest IP standards required ≥576. Usually 1492 is safe, often 1500, sometimes more.)

The page you're downloading from `pqcrypto.org` doesn't fit.

Browser actually makes "TCP connection" to `pqcrypto.org`. Inside that connection: sends HTTP request, receives response.

Browser → server:
“`SYN 168bb5d9`”

Server → browser:
“`ACK 168bb5da, SYN 747bfa41`”

Browser → server:
“`ACK 747bfa42`”

Server now allocates buffers for this TCP connection.

Browser splits data into packets, counting bytes from `168bb5da`.

Server splits data into packets, counting bytes from `747bfa42`.

Main feature advertised by TCP:
“reliable data streams”.

Internet sometimes loses packets
or delivers packets out of order.
Doesn’t confuse TCP connections:
computer checks the counter
inside each TCP packet.

Computer retransmits data
if data is not acknowledged.
Complicated rules to decide
retransmission schedule,
avoiding network congestion.

Stream-level crypto

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

`https://www.pqcrypto.org`
uses HTTP over TLS over TCP.

Your browser
- finds address `131.155.70.11`;
- makes TCP connection;
- inside the TCP connection, builds a TLS connection by exchanging crypto keys;
- inside the TLS connection, sends HTTP request etc.

What happens if attacker
forges a DNS packet
pointing to fake server?
Or a TCP packet
with bogus data?

DNS software is fooled.
TCP software is fooled.
TLS software sees that
something has gone wrong,
but has no way to recover.

Browser using TLS can
make a whole new connection,
but this is slow and fragile.
Huge damage from forged packet.

Modern trend (e.g., DNSCurve, CurveCP; see also MinimaLT, Google's QUIC): Authenticate and encrypt each packet separately.

Discard forged packet immediately: no damage.
Retransmit packet if no *authenticated* acknowledgment.

Modern trend (e.g., DNSCurve, CurveCP; see also MinimaLT, Google's QUIC): Authenticate and encrypt each packet separately.

Discard forged packet immediately: no damage. Retransmit packet if no *authenticated* acknowledgment.

Engineering advantage: Packet-level crypto works for more protocols than stream-level crypto.

Modern trend (e.g., DNSCurve, CurveCP; see also MinimaLT, Google's QUIC): Authenticate and encrypt each packet separately.

Discard forged packet immediately: no damage.
Retransmit packet if no *authenticated* acknowledgment.

Engineering advantage:
Packet-level crypto works for more protocols than stream-level crypto.

Disadvantage:
Crypto must fit into packet.

The KEM+AE philosophy

Original view of RSA:
Message m is encrypted
as $m^e \bmod pq$.

The KEM+AE philosophy

Original view of RSA:
Message m is encrypted
as m^e mod pq.

"Hybrid" view of RSA,
including random padding:
Choose random AES-GCM key k.
Randomly pad k as r.
Encrypt r as r^e mod pq.
Encrypt m under k.

The KEM+AE philosophy

Original view of RSA:
Message m is encrypted
as $m^e \bmod pq$.

“Hybrid” view of RSA,
including random padding:
Choose random AES-GCM key k.
Randomly pad k as r.
Encrypt r as $r^e \bmod pq$.
Encrypt m under k.

Fragile, many problems:
e.g., Coppersmith attack,
Bleichenbacher attack,
bogus OAEP security proof.

Shoup's "KEM+DEM" view:

"Key encapsulation mechanism":
Choose random r mod pq.
Encrypt r as r^e mod pq.
Define $k = H(r, r^e \bmod pq)$.

"Data encapsulation mechanism":
Encrypt and authenticate
m under AES-GCM key k.

Authenticator catches
any modification of r^e mod pq.

Much easier to get right.
Also generalizes nicely.
Can mix multiple hashes.

DEM security hypothesis:
weak single-message version
of security for secret-key
authenticated encryption.

Chou: Is it safe to reuse k
for multiple messages?

Answer: KEM+AE is safe;
KEM+AE ⇒ KEM+“nDEM”.
(But need literature on this!)
AES-GCM, Salsa20-Poly1305, etc.
aim for full AE security goal.

More complicated alternative:
Use KEM+DEM to encrypt an
n-time secret key m; reuse m.

DNSCurve: ECDH for DNS

Server knows ECDH secret key s.

Client knows ECDH secret key c, server's public key $S = sG$.

Client $\to$ server:
packet containing $cG, E_k(0, q)$
where $k = H(cS)$;
E is authenticated cipher;
q is DNS query.

Server $\to$ client:
packet containing $E_k(1, r)$
where r is DNS response.

Client can reuse c
across multiple queries,
but this leaks metadata.
Let's assume one-time c.

Client can reuse c
across multiple queries,
but this leaks metadata.
Let's assume one-time c.

KEM+AE view:

Client is sending $k = H(cS)$
encapsulated as cG.
This is an "ECDH KEM".

Client can reuse c
across multiple queries,
but this leaks metadata.
Let's assume one-time c.

KEM+AE view:

Client is sending $k = H(cS)$
encapsulated as cG.
This is an "ECDH KEM".

Client then uses k
to authenticate+encrypt.

Server also uses k
to authenticate+encrypt.

Post-quantum encrypted DNS

"McEliece KEM":
Client sends $k = H(c, e, Sc + e)$
encapsulated as $Sc + e$.

Random $c \in \mathbf{F}_2^{5413}$;
random small $e \in \mathbf{F}_2^{6960}$;
public key $S \in \mathbf{F}_2^{6960 \times 5413}$.

Post-quantum encrypted DNS

"McEliece KEM":
Client sends $k = H(c, e, Sc + e)$
encapsulated as $Sc + e$.

Random $c \in \mathbf{F}_2^{5413}$;
random small $e \in \mathbf{F}_2^{6960}$;
public key $S \in \mathbf{F}_2^{6960\times 5413}$.

S has secret Goppa structure
allowing server to decrypt.

Post-quantum encrypted DNS

"McEliece KEM":
Client sends $k = H(c, e, Sc + e)$
encapsulated as $Sc + e$.

Random $c \in \mathbf{F}_2^{5413}$;
random small $e \in \mathbf{F}_2^{6960}$;
public key $S \in \mathbf{F}_2^{6960\times 5413}$.

S has secret Goppa structure
allowing server to decrypt.

"Niederreiter KEM", smaller:
Client sends $k = H(e, S'e)$
encapsulated as $S'e \in \mathbf{F}_2^{1547}$.

Client → server:
packet containing $Sc + e, E_k(0, q)$.
(Combine with ECDH KEM.)

Server → client:
packet containing $E_k(1, r)$.

Client → server:
packet containing $Sc+e, E_k(0,q)$.
(Combine with ECDH KEM.)

Server → client:
packet containing $E_k(1,r)$.

r states a server address
and the server's public key.
What if the key is too long
to fit into a single packet?

One simple answer:
Client separately requests
each block of public key.
Can do many requests in parallel.

Confidentiality:
Attacker can't guess k,
can't decrypt $E_k(0, q), E_k(1, r)$.

Integrity:
Server never signs anything,
but E_k includes authentication.
Attacker can send new queries
but can't forge q or r.
Attacker *can* replay request.

Availability:
Client discards forgery,
continues waiting for reply,
eventually retransmits request.

Big keys

McEliece public key is 1MB for long-term confidence today.

Is this size a problem? Do we need to switch to lower-confidence approaches such as NTRU or QC-MDPC?

Size of average web page in Alexa Top 1000000: 1.8MB.

Web page often needs public keys for several servers, but public key for a server can be reused for many pages.

Most important limitation
on reuse of public keys:
switching to new keys
and **promptly erasing old keys**.

Rationale: “forward secrecy”—
subsequent theft of computer
doesn’t allow decryption.

e.g. Microsoft SChannel
switches keys every two hours.

Safer: new key every minute.

Easier to implement:
new key every connection.

What is the performance of a new key every minute?

If server makes new key:
key gen, ≤ 1 per minute;
client encrypts to new key;
server decrypts.

What is the performance of
a new key every minute?

If server makes new key:
key gen, ≤ 1 per minute;
client encrypts to new key;
server decrypts.

If client makes new key:
client has key-gen cost;
server has encryption cost;
client has decryption cost.

Either way:
one key transmission for each
active client-server pair.

How does a *stateless* server encrypt to a new client key without storing the key?

How does a *stateless* server encrypt to a new client key without storing the key?

Slice McEliece public key so that each slice of encryption produces separate small output.

Client sends slices (in parallel), receives outputs as cookies, sends cookies (in parallel).
Server combines cookies.
Continue up through tree.

Server generates randomness as secret function of key hash.
Statelessly verifies key hash.