The post-quantum Internet

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

Includes joint work with:

Tanja Lange

Technische Universiteit Eindhoven

Risk management

"Combining congruences": state-of-the-art pre-quantum attack against original DH, RSA, and some lattice systems.

Long history, including many major improvements:
1975, CFRAC;
1977, linear sieve (LS);
1982, quadratic sieve (QS);
1990, number-field sieve (NFS);
1994, function-field sieve (FFS);
2006, medium-prime FFS/NFS;
2013, $x^q - x$ FFS.

Also many smaller improvements: $>$100 scientific papers.

Costs of these algorithms for breaking RSA-1024, RSA-2048:
$\approx 2^{120}$, $\approx 2^{170}$, CFRAC;
$\approx 2^{110}$, $\approx 2^{160}$, LS;
$\approx 2^{100}$, $\approx 2^{150}$, QS;
$\approx 2^{80}$, $\approx 2^{112}$, NFS.
(FFS is not relevant to RSA.)

Also many smaller improvements: $>$100 scientific papers.

Costs of these algorithms for breaking RSA-1024, RSA-2048:
$\approx 2^{120}$, $\approx 2^{170}$, CFRAC;
$\approx 2^{110}$, $\approx 2^{160}$, LS;
$\approx 2^{100}$, $\approx 2^{150}$, QS;
$\approx 2^{80}$, $\approx 2^{112}$, NFS.
(FFS is not relevant to RSA.)

How much risk is there of future breakthroughs?

Also many smaller improvements:
$>$100 scientific papers.

Costs of these algorithms for
breaking RSA-1024, RSA-2048:
$\approx 2^{120}$, $\approx 2^{170}$, CFRAC;
$\approx 2^{110}$, $\approx 2^{160}$, LS;
$\approx 2^{100}$, $\approx 2^{150}$, QS;
$\approx 2^{80}$, $\approx 2^{112}$, NFS.
(FFS is not relevant to RSA.)

How much risk is there
of future breakthroughs?

How much risk is there
of secret breakthroughs?

If we put enough effort into exploring Attack Mountain, will we find the highest peak? At least within ϵ?

If we put enough effort into
exploring Attack Mountain,
will we find the highest peak?
At least within ϵ?

Combining-Congruences Mountain
is a huge, foggy, high-dimensional
mountain with many paths up.
Scary: easy to imagine that
we're not at the top yet.

If we put enough effort into exploring Attack Mountain, will we find the highest peak? At least within ϵ?

Combining-Congruences Mountain is a huge, foggy, high-dimensional mountain with many paths up. Scary: easy to imagine that we're not at the top yet.

18-year bet announced in 2014: Joux wins if RSA-2048 is broken first by pre-quantum algorithms; I win if RSA-2048 is broken first by quantum algorithms.

Conservative cryptographers
prefer mountains that seem
less huge, less foggy,
more thoroughly explored.

Conservative cryptographers prefer mountains that seem less huge, less foggy, more thoroughly explored.

1986 Miller "Use of elliptic curves in cryptography": "It is extremely unlikely that an 'index calculus' attack [combining-congruences attack] on the elliptic curve method will ever be able to work."

Conservative cryptographers
prefer mountains that seem
less huge, less foggy,
more thoroughly explored.

1986 Miller “Use of elliptic curves in cryptography”: “It is extremely unlikely that an ‘index calculus’ attack [combining-congruences attack] on the elliptic curve method will ever be able to work.”

This is the core argument for ECC. Exceptions: rare curves with special structure—e.g., pairings.

2015 Lange: “Would you bet your kidneys on that?”

The setting

It's 2050. Quantum computers were built years ago.

Evil Party A now runs the country and has access to records of practically all 21st-century Internet traffic. Evil Party A thinks vaccinations are bad and jails anybody who was vaccinated during the past 70 years. Doctor-patient confidentiality is still protected by law, but your health record from birth has been online since 2020. Your health record is protected only by encryption to your doctor's public key, using our recommendation from 2015 of public-key and authenticated symmetric encryption.

Organs are a scarce resource. Hospitals pay high prices for organs if they can identify the donor (DNA tests are cheap) and are presented with the donor's digitally signed Donor Volunteer Statement. They use our 2015 recommended signature system.

(This is meant to scare you, so that you recommend only what you trust. Let's make sure that this dystopia will not happen.)

Risk of future attacker having
big universal quantum computer:
noticeable probability;
terrifying impact.

Risk of future attacker having big universal quantum computer: noticeable probability; terrifying impact.

Fortunately, we already know some confidence-inspiring post-quantum systems, including

- hash-based signatures;
- McEliece public-key encryption;
- AES-256 etc.

https://pqcrypto.eu.org/docs/initial-recommendations.pdf

Application: software updates

Your computer downloads new version of its OS.

Your computer checks signature on the download from the OS manufacturer.

Critical use of crypto! Otherwise criminals could insert malware into the OS.

e.g. OpenBSD updates are signed using state-of-the-art ECC signature system: Ed25519.

Pre-quantum signature system P
needs to be replaced with
post-quantum signature system Q.

Pre-quantum signature system P needs to be replaced with post-quantum signature system Q.

Make auditors happier:
Replace P with $P + Q$.

$P + Q$ public key concatenates P public key, Q public key.
$P + Q$ signature concatenates P signature, Q signature.

Pre-quantum signature system P needs to be replaced with post-quantum signature system Q.

Make auditors happier:
Replace P with $P+Q$.

$P+Q$ public key concatenates P public key, Q public key.
$P+Q$ signature concatenates P signature, Q signature.

Want a tiny public key?
Replace public key with hash.
Include missing information ($\leq$ entire key) inside signature.

e.g. Ed25519+SPHINCS-256.

SPHINCS-256 signature is 41KB;
≈50 million cycles to generate;
≈1 million cycles to verify.
Negligible cost to sign, transmit, verify compared to OS update.

e.g. Ed25519+SPHINCS-256.

SPHINCS-256 signature is 41KB;
≈50 million cycles to generate;
≈1 million cycles to verify.
Negligible cost to sign, transmit,
verify compared to OS update.

+Ed25519: unnoticeable cost.
Some extra system complexity,
but the system includes
Ed25519 code anyway.

e.g. Ed25519+SPHINCS-256.

SPHINCS-256 signature is 41KB; $\approx$50 million cycles to generate; $\approx$1 million cycles to verify. Negligible cost to sign, transmit, verify compared to OS update.

+Ed25519: unnoticeable cost. Some extra system complexity, but the system includes Ed25519 code anyway.

Auditor sees very easily that Ed25519+SPHINCS-256 security $\geq$ Ed25519 security.

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

Pre-quantum situation:
Hash-based signatures are
even more confidence-inspiring
than ECC signatures.
But understanding this fact
takes extra work for auditor.

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

Pre-quantum situation:
Hash-based signatures are
even more confidence-inspiring
than ECC signatures.
But understanding this fact
takes extra work for auditor.

Long-term situation:
Users see quantum computers
easily breaking P. Simplify system
by switching from $P + Q$ to Q.

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.
$P + Q$: hash concatenation.

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

Server → 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 $S'e \in \mathbf{F}_2^{1547}$.

“NTRU KEM”,
obviously totally unrelated:
Client sends $k = H(c, e, Sc + e)$
encapsulated as $Sc + e$.

“NTRU KEM”,
obviously totally unrelated:
Client sends $k = H(c, e, Sc + e)$
encapsulated as $Sc + e$.

Random small
$c, e \in (\mathbf{Z}/q)[x]/(x^n - 1)$;
public key $S \in (\mathbf{Z}/q)[x]/(x^n - 1)$.

Secretly $S = 3s/t$; small s, t.
Server recovers $3sc + te$,
then te mod 3, then e, then c.

"NTRU KEM",
obviously totally unrelated:
Client sends $k = H(c, e, Sc + e)$
encapsulated as $Sc + e$.

Random small
$c, e \in (\mathbf{Z}/q)[x]/(x^n - 1)$;
public key $S \in (\mathbf{Z}/q)[x]/(x^n - 1)$.

Secretly $S = 3s/t$; small s, t.
Server recovers $3sc + te$,
then te mod 3, then e, then c.

Can imitate Niederreiter in the NTRU context: e.g. "Ring-LWR".

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.

Cookies

What if $E_k(0, q)$ doesn't fit into same packet as $Sc + e$?

Client sends short $E_k(0, q')$ containing a **cookie request** q'.

Server sends $E_k(1, r')$ containing **cookie** r': server state (including k) encrypted from server to itself. Server can now forget state.

Client sends packet $r', E_k(2, q)$. Server recovers state, decrypts.

Server sends $E_k(3, r)$.

Client authentication

Same strategy works
for protecting connections.
$C \to S$, $S \to C$ data flow
isn't special; reuse k for
many packets each direction.

Client authentication

Same strategy works
for protecting connections.
$C \to S$, $S \to C$ data flow
isn't special; reuse k for
many packets each direction.

Another TCP availability problem:
server allocates buffers for each
connection; runs out of memory.

Client authentication

Same strategy works for protecting connections. $C \to S$, $S \to C$ data flow isn't special; reuse k for many packets each direction.

Another TCP availability problem: server allocates buffers for each connection; runs out of memory.

Semi-solution: Allocate buffers only after client sends r'.

Client authentication

Same strategy works for protecting connections. $C \to S$, $S \to C$ data flow isn't special; reuse k for many packets each direction.

Another TCP availability problem: server allocates buffers for each connection; runs out of memory.

Semi-solution: Allocate buffers only after client sends r'.

Solution 1: Hashcash from client.

Solution 2: Redo protocols to avoid state on server.

Imitate NFS, not HTTP.

Solution 2: Redo protocols to avoid state on server.

Imitate NFS, not HTTP.

Solution 3 for, e.g., SSH: Authenticate client.

Server can authenticate client without signatures, same way client authenticates server:

- Send to client's public key encapsulation of new key k'.
- Hash k' into shared secret.

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.