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.
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?”
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 → 131.155.70.11:
“Where is www.pqcrypto.org?”
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?”
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?”
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?”

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

131.155.71.143 → browser: “131.155.70.11”
Internet Protocol

Internet Protocol communicates “packets”: limited-length byte strings.

Each computer on the Internet has a 4-byte “IP address”.

For example, 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?”

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

131.155.71.143 → browser: “131.155.70.11”
**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?”  
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”.
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?”

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.”
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”
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 “199.19.54.1”, the .org server address, by asking the root name server.
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 \(\rightarrow\) 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 \(\rightarrow\) browser: “131.155.70.11”

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

Browser \(\rightarrow\) 199.19.54.1: “Where is www.pqcrypto.org?”

199.19.54.1 \(\rightarrow\) 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.
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?”

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 ≥ 576. Usually 1492 is safe,
often 1500, sometimes more.)
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?"

131.155.71.143 → 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 ≥ 576. Usually 1492 is safe, often 1500, sometimes more.)
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?"

131.155.71.143 → browser:
"131.155.70.11"

TCP: Transmission Control Protocol

Packets are limited to 1280 bytes.

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

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.
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 ≥576. Usually 1492 is safe, often 1500, sometimes more.)
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.)

The page you’re downloading from pqcrypto.org doesn’t fit.
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 ≥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 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 ≥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.

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.
The name-server address, "131.155.71.143", is learned by asking the .org name server.

Browser \( \rightarrow \) 199.19.54.1: "Where is www.pqcrypto.org?"

199.19.54.1 \( \rightarrow \) 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.)

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 \( \rightarrow \) server: “SYN 168bb5d9”

Server \( \rightarrow \) browser: “ACK 168bb5da, SYN 747bfa41”

Browser \( \rightarrow \) server: “ACK 747bfa42”

Server now allocates buffers for this TCP connection.

Browser splits data into packets, counting bytes from 168bb5d9.

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

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.
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.

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.
Transmission Control Protocol (TCP) packets are limited to 1280 bytes. The limit actually depends on network. Older IP standards required packets of \( \geq 576 \) bytes. Usually 1492 bytes is safe, often 1500 bytes, 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.
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 if data is not acknowledged. Complicated rules to decide retransmission schedule, avoiding network congestion.
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.
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.
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.
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.
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.
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.
Main feature advertised by TCP: “reliable data streams”. Internet sometimes loses packets or delivers packets out of order. Does not 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.
Main feature advertised by TCP: "reliable data streams".

Internet sometimes loses packets or delivers packets out of order. Does'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.
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.
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.
Stream-level crypto

http://www.pqcrypto.org
TP over TCP.

https://www.pqcrypto.org
TP over TLS over TCP.

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.
Stream-level crypto 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.
Stream-level crypto 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.
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.
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.

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

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

Disadvantage: Crypto must fit into packet.
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.

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 \( m^e \mod pq \).
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.

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 \( m^e \mod pq \).
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 $m^e \mod pq$. 
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 $m^e \mod pq$. 
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 $m^e \mod pq$.

“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 $k$. 
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 $m^e \mod pq$.

“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 $k$.

Fragile, many problems:
e.g., Coppersmith attack, Bleichenbacher attack, bogus OAEP security proof.
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 $m^e \mod pq$.

“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 $k$.

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 $k = H(r; r^e \mod pq)$.

“Data encapsulation mechanism”: Encrypt and authenticate $m$ under $k$. Authenticator catches any modification of $r^e \mod pq$.

Much easier to get right. Also generalizes nicely. Can mix multiple hashes.
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 \( m^e \mod pq \).

“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 \( k \).

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 \( k = H(r, r^e) \).

“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.
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 \( m^e \text{ mod } pq \).

“Hybrid” view of RSA, including random padding: Choose random AES-GCM key \( k \). Randomly pad \( k \) as \( r \). Encrypt \( r \) as \( r^e \text{ mod } pq \). 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 \( r \text{ mod } pq \). Encrypt \( r \) as \( r^e \text{ mod } pq \). Define \( k = H(r, r^e \text{ mod } pq) \).

“Data encapsulation mechanism”: Encrypt and authenticate \( m \) under AES-GCM key \( k \).

Authenticator catches any modification of \( r^e \text{ mod } pq \).

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

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 $k = H(r, r^e \mod pq)$.

“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.
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 $k$.

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 $k = H(r, r^e \mod pq)$.

"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 $\Rightarrow$ KEM+"n"DEM.
(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 $m$.

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 $k$.

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 $k = H(r, r^e \mod pq)$.

“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 $\Rightarrow$ KEM+AE
(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 $m$.

Shoup’s “KEM+DEM” view:

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

“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 $\Rightarrow$ KEM+“$n$DEM”.
(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 $r$. 
Shoup’s “KEM+DEM” view:

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

“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 $\Rightarrow$ 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 $m$. 
“KEM+DEM” view:
"Key encapsulation mechanism": Choose random $r \mod pq$.
Encrypt $r$ as $r^e \mod pq$.
$k = H(r, r^e \mod pq)$.
"Data encapsulation mechanism": Encrypt and authenticate $m$ under AES-GCM key $k$.
Authenticator catches 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 $\Rightarrow$ 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 $\rightarrow$ server: packet containing $cG$; $E_k(0; q)$ where $k = H(cS)$;
$E$ is authenticated cipher; $q$ is DNS query.
Server $\rightarrow$ client: packet containing $E_k(1; r)$ where $r$ is DNS response.
Shoup's "KEM+DEM" view:

"Key encapsulation mechanism":
Choose random $r \mod pq$.
Encrypt $r$ as $r^e \mod pq$.

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

Server $\rightarrow$ client:
packet containing $E_k(1; r)$ where $r$ is DNS response.
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 $\rightarrow$ server:
packet containing $cG, E_k(0, q)$ where $k = H(cS)$; $E$ is authenticated cipher; $q$ is DNS query.

Server $\rightarrow$ client:
packet containing $E_k(1, r)$ where $r$ is DNS response.
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 $\rightarrow$ server:
packet containing $cG, E_k(0, q)$
where $k = H(cS)$;
$E$ is authenticated cipher;
$q$ is DNS query.

Server $\rightarrow$ client:
packet containing $E_k(1, r)$
where $r$ is DNS response.
DEM security hypothesis: weak single-message version of security for secret-key authenticated encryption.

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

- KEM+AE is safe; $AE \Rightarrow KEM+"nDEM"$. (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 $c$. 
DEMS security hypothesis: weak single-message version of security for secret-key authenticated encryption.

Chou: Is it safe to reuse $k$ for multiple messages? Answer: $\text{KEM+AE}$ is safe; $\text{KEM+AE} \Rightarrow \text{KEM+"nDEM"}. (\text{But need literature on this!})$

AES-GCM, Salsa20-Poly1305, etc. aim for full AE security goal.

More complicated alternative: Use $\text{KEM+DEM}$ 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 $\rightarrow$ server:
packet containing $cG, E_k(0, q)$ where $k = H(cS)$; $E$ is authenticated cipher; $q$ is DNS query.

Server $\rightarrow$ 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 $c$...
DNSCurve: ECDH for DNS

Server knows ECDH secret key $s$.

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

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

Server $\rightarrow$ 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 $c$. 

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+$^n$ DEM.
(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 $m$. 

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

DNSCurve: ECDH for DNS

Server knows ECDH secret key $s$.

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

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

Server $\rightarrow$ client:
packet containing $E_k(1, r)$
where $r$ is DNS response.
**DNSCurve: ECDH for DNS**

Server knows ECDH secret key $s$.

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

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

Server $\rightarrow$ 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 $c$. 
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 $c$.

KEM+AE view:
Client is sending $k = H(cS)$ encapsulated as $cG$.
This is an “ECDH KEM”. 
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 $c$.

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.
**DNSCurve: ECDH for DNS**

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 $c \in F_{5413}^2$; random small $e \in F_{6960}^2$; public key $S \in F_{6960} \times F_{5413}^2$. 

  Client knows ECDH secret key $s$.

  Server knows ECDH secret key $c$, server’s public key $S = sG$.

  **Client → server:**

  Packet containing $cG, E_k(0, q)$ where $k = H(cS)$; authenticated cipher; $S$ 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 $c$.

**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 \( c \in \mathbb{F}_{5413}^2 \); random small \( e \in \mathbb{F}_{6960}^2 \); public key \( S \in \mathbb{F}_{6960} \times \mathbb{F}_{5413}^2 \).
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 $c$.

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 $c \in F_{5413}^2$; random small $e \in F_{6960}^2$; public key $S \in F_{6960 \times 5413}^2$. 
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 $c \in \mathbb{F}_2^{5413}$;
random small $e \in \mathbb{F}_2^{6960}$;
public key $S \in \mathbb{F}_2^{6960 \times 5413}$.
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 $c \in \mathbb{F}_2^{5413}$;
random small $e \in \mathbb{F}_2^{6960}$;
public key $S \in \mathbb{F}_2^{6960\times5413}$.

$S$ has secret Goppa structure allowing server to decrypt.
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 $c \in \mathbb{F}_2^{5413}$;
random small $e \in \mathbb{F}_2^{6960}$;
public key $S \in \mathbb{F}_2^{6960 \times 5413}$.

$S$ has secret Goppa structure allowing server to decrypt.

“Niederreiter KEM”, smaller:
Client sends $k = H(e, S'e)$ encapsulated as $S'e \in \mathbb{F}_2^{1547}$.
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 $c \in F^{5413}_2$; random small $e \in F^{6960}_2$; public key $S \in F^{6960} \times F^{5413}_2$.
$S$ has secret Goppa structure allowing server to decrypt.

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

Client → server:
packet containing $Sc + e; E^k(0; q)$.
(Combine with ECDH KEM.)

Server → client:
packet containing $E^k(1; r)$.

Server
(Combine with ECDH KEM)
Post-quantum encrypted DNS

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

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

$S$ has secret Goppa structure allowing server to decrypt.

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

Client $\rightarrow$ server: 
packet containing $Sc + e; E^k(0; q)$.
(Combine with ECDH KEM.)

Server $\rightarrow$ client: 
packet containing $E^k(1 ; r)$. 

Post-quantum encrypted DNS

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

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

$S$ has secret Goppa structure allowing server to decrypt.

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

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

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

Post-quantum encrypted DNS

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

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

$S$ has secret Goppa structure allowing server to decrypt.

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

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

Server $\rightarrow$ client:
packet containing $E_k(1, r)$. 
Post-quantum encrypted DNS

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

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

$S$ has secret Goppa structure
allowing server to decrypt.

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

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

Server $\rightarrow$ 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.
Post-quantum encrypted DNS

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

- Random $c \in \mathbb{F}_{5413}^2$;
- Small $e \in \mathbb{F}_{6960}^2$;
- Public key $S \in \mathbb{F}_{6960}^{5413}$.

Secret Goppa structure allows server to decrypt.

**Niederreiter KEM**, smaller: Client sends $k = H(e, S' e)$ encapsulated as $S' e \in \mathbb{F}_{1547}^2$.

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 $r$.

Availability: Attacker can replay request.

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 decrypt.

Availability: Client discards forgery, continues waiting for reply, eventually retransmits request.
Post-quantum encrypted DNS

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

Random $c \in \mathbb{F}_{6960}^2$;
random small $e \in \mathbb{F}_{5413}^2$;
public key $S \in \mathbb{F}_{6960}^{5413}$.
$S$ has secret Goppa structure
allowing server to decrypt.

“Niederreiter KEM”, smaller:
Client sends $k = H(e, S'e)$
encapsulated as $S'e \in \mathbb{F}_{1547}^2$.

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 $E_k(0, q)$.

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.
Post-quantum encrypted DNS

“McEliece KEM”:
Client sends $k = H(c; e; Sc + e)$
encapsulated as $Sc + e$.

Random $c \in F_{5413}^2$;
random small $e \in F_{6960}^2$;
public key $S \in F_{6960} \times F_{5413}^2$.

$S$ has secret Goppa structure
allowing server to decrypt.

“Niederreiter KEM”, smaller:
Client sends $k = H(e; S' e)$
encapsulated as $S' e \in F_{1547}^2$.

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.
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.
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?

A 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 an average web page
in Alexa Top 100,000: 1.8MB.
Web page often needs
public keys for several servers,
but public key for a server
can be reused for many pages.
\[ Sc + e, E_k(0, q). \]
(Combine with ECDH KEM.)

\[ E_k(1, r). \]

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
and can't 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 \textit{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
Client → server: packet containing $S + e; E_k(0; q)$. (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.
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.
Confidentiality:
Attacker can't guess $k$, can't decrypt $E_k(0, q), E_k(1, r)$.

Integrity:
Server never signs anything, 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.
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.
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.
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.
The McEliece public key is 1MB in size. Is this size a problem for long-term confidence today? Do we need to switch to lower-confidence approaches such as NTRU or QC-MDPC? The average web page in Alexa Top 1000000 is 1.8MB. A page often needs public keys for several servers, but a 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.
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.
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, \( \leq 1 \) per minute;

client encrypts to new key;

server decrypts.
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, \(\leq 1\) per minute; client encrypts to new key; server decrypts.
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, \( \leq 1 \) 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.
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.

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, $\leq 1$ 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?
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, $\leq 1$ 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?
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, $\leq 1$ 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?
What is the performance of a new key every minute?

If server makes new key:
key gen, \(\leq 1\) 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?
What is the performance of a new key every minute? If server makes new key: key gen, \( \leq 1 \) 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? 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.