TLS

Benchmark symmetric ciphers

Symmetric ciphers encrypt and decrypt bulk data. TLS 1.3 allows two families: **AES-GCM** and **ChaCha20-Poly1305**.

Run the following benchmarks:

```
openssl speed -evp aes-128-gcm
openssl speed -evp aes-256-gcm
openssl speed -evp chacha20-poly1305
```

Each command runs for a few seconds and reports throughput in bytes per second for different block sizes.

Answer:

- What throughput (in MB/s) does your machine achieve for AES-128-GCM with 8192-byte blocks?
- How does AES-256-GCM compare to AES-128-GCM?
- How does ChaCha20-Poly1305 compare to AES-GCM?

> Hint
>
> You can check whether your CPU has hardware AES acceleration:
>
> ```
> # Linux
> grep -o aes /proc/cpuinfo | head -1
>
> # macOS
> sysctl hw.optional.aes
> ```

Benchmark hash functions

Hash functions are used in TLS for message authentication (HMAC) and in the key derivation process (HKDF). TLS 1.3 uses SHA-256 or SHA-384.

Run:

```
openssl speed -evp sha256
openssl speed -evp sha384
```

Answer:

- What throughput does SHA-256 achieve on large blocks (8192 bytes)?
- How does SHA-384 compare to SHA-256?

Benchmark RSA signatures

RSA is used in TLS for digital signatures during the handshake. These operations are much slower than symmetric encryption — which is why they are only used once per connection, not for every byte of data.

The script `tls/rsa-benchmark.sh` benchmarks RSA for four key sizes (1024, 2048, 3072, 4096 bits) and reports sign/verify operations per second:

```
./tls/rsa-benchmark.sh
```

You can also run individual key sizes directly:

```
openssl speed rsa2048
openssl speed rsa4096
```

Answer:

- How many RSA-2048 signatures per second can your machine compute?
- How many verifications per second?
- Is signing faster or slower than verifying? Why?
- How does RSA-4096 compare to RSA-2048?
- How does the performance scale as the key size doubles?

Benchmark elliptic-curve operations

Elliptic-curve cryptography provides equivalent security to RSA but with much smaller keys. TLS 1.3 uses ECDSA for signatures and ECDHE for key exchange.

Run:

```
openssl speed ecdsap256
openssl speed ecdhx25519
openssl speed ecdhp256
```

Answer:

- How do ECDSA P-256 sign/verify speeds compare to RSA-2048?
- How many X25519 key exchanges per second can your machine perform?
- How does ECDH P-256 compare to X25519?

Benchmark Diffie-Hellman key exchange

The script `tls/dh-benchmark.py` demonstrates the classic Diffie-Hellman key exchange and benchmarks it with different key sizes.

First, read the step-by-step demonstration at the top of the output. It uses small numbers (p=23, g=5) so you can verify the computation by hand.

Run the script:

```
python3 tls/dh-benchmark.py
```

The benchmark tests key sizes from 64 bits (toy, insecure) up to 4096 bits (RFC 7919 standard). The standard primes come from:

- 1024-bit: RFC 2409 Section 6.2 (Second Oakley Group)
- 2048-bit: RFC 7919 Appendix A.1 (ffdhe2048)
- 4096-bit: RFC 7919 Appendix A.3 (ffdhe4096)

Answer:

- How does the computation time grow as the key size doubles? Is it linear, quadratic, or worse?

- Compare the DH-2048 throughput (exchanges/s) with the ECDH X25519 result from the elliptic-curve exercise above. Both provide roughly 128-bit security — which is faster?
- Why do you think TLS 1.3 defaults to X25519 for key exchange?

Comparing symmetric and asymmetric performance

Using your results from the previous exercises, fill in a table like this:

Algorithm	Operation	Your measurement
AES-128-GCM	Encrypt (8192 B blocks)	… MB/s
AES-256-GCM	Encrypt (8192 B blocks)	… MB/s
ChaCha20-Poly1305	Encrypt (8192 B blocks)	… MB/s
SHA-256	Hash (8192 B blocks)	… MB/s
RSA-2048	Sign / Verify	… / … ops/s
RSA-4096	Sign / Verify	… / … ops/s
ECDSA P-256	Sign / Verify	… / … ops/s
ECDH X25519	Key exchange	… ops/s
DH-2048	Key exchange	… ops/s
DH-4096	Key exchange	… ops/s

Answer:

1. If you were encrypting a 1 GB file, roughly how long would it take with AES-128-GCM? (Divide 1 GB by your measured throughput.)
2. How many TLS handshakes per second could your machine sustain if the bottleneck were the RSA-2048 signature? What about ECDSA P-256?
3. Why does TLS use asymmetric cryptography **only during the handshake** and switch to symmetric cryptography for the actual data transfer?
4. TLS 1.3 removed RSA key exchange (only ECDHE is allowed). Based on your measurements, what is the performance difference? What security benefit does ECDHE provide that static RSA key exchange does not?

> Hint
>
> The security benefit is called **forward secrecy**: even if the server's private key is compromised in the future, past recorded sessions cannot be decrypted, because the ephemeral DH keys were discarded after the handshake.

Capture and decrypt the TLS handshake

Restart the server, then use the client with PCAP capture and key logging:

```
python3 tls/simple-server.py --cert server-cert.pem --key server-key.pem
```

In another terminal:

```
sudo python3 tls/tls-client.py localhost -p 8443 --no-cert-verify \
    --pcap tls-handshake.pcap --keylog tls.keys
```

Open the capture in Wireshark:

```
wireshark tls-handshake.pcap
```

To decrypt the TLS traffic, configure the key log file in Wireshark:

1. Go to `Edit` → `Preferences` → `Protocols` → `TLS`
2. Set **(Pre)-Master-Secret log filename** to the full path of `tls.keys`
3. Click OK

Useful Wireshark display filters for TLS

Filter	Description
`tls`	Show all TLS packets
`tls.handshake`	Show only handshake messages
`tls.handshake.type == 1`	ClientHello
`tls.handshake.type == 2`	ServerHello
`tls.handshake.type == 11`	Certificate
`tls.handshake.type == 20`	Finished
`tls.record.content_type == 23`	Application data (encrypted)

In the Wireshark trace, answer:

- How many packets does the TLS 1.3 handshake take? (Count from ClientHello to the first application data.)
- In the **ClientHello**, what cipher suites does the client propose? What key exchange groups does it offer?
- In the **ServerHello**, which cipher suite and key exchange group did the server choose?
- Can you find the server's **Certificate** message? (With TLS 1.3 it is encrypted — you need the key log file to see it.)
- After configuring the key log file, can you see the decrypted "Hello" message sent by the server?

TLS session resumption

Establishing a new TLS 1.3 connection requires a full handshake: key exchange, signature verification, and certificate transmission. This adds latency — at least one extra round trip before the first byte of application data.

To speed up subsequent connections, TLS 1.3 supports **session resumption** using Pre-Shared Keys (PSK). After a successful handshake, the server sends a **NewSessionTicket** message containing an encrypted token. On the next connection, the client includes this token in a `pre_shared_key` extension in the ClientHello. The server recognises it and skips the certificate exchange.

The script `tls/tls-resume.py` demonstrates this mechanism. It saves the session ticket to a file after the first connection and reuses it on subsequent connections.

Step 1 — First connection (full handshake). Delete any old session file to start fresh:

```
rm -f tls.session
python3 tls/tls-resume.py www.computer-networking.info --keylog tls-full.keys
```

Observe the output. The script saves the session to `tls.session`.

Step 2 — Second connection (resumed handshake). Run the same command again — the script loads the saved session and attempts resumption:

```
python3 tls/tls-resume.py www.computer-networking.info --keylog tls-resume.keys
```

Observe the output and compare with step 1.

Step 3 — Capture both connections. Start tshark, then run the two connections in sequence:

```
sudo tshark -i any -p -w tls-resumption.pcap -f "tcp port 443" &
sleep 2

rm -f tls.session
python3 tls/tls-resume.py www.computer-networking.info --keylog tls-resumption.keys
sleep 1
python3 tls/tls-resume.py www.computer-networking.info --keylog tls-resumption.keys
sleep 1

sudo kill %1
```

Open the capture in Wireshark with the key log file:

```
wireshark tls-resumption.pcap
```

Configure `Edit → Preferences → Protocols → TLS → (Pre)-Master-Secret log filename` to point to `tls-resumption.keys`.

In the trace you should see **two** TLS connections. Compare them:

1. In the **first** ClientHello, is there a `pre_shared_key` extension?
2. In the **second** ClientHello, is there a `pre_shared_key` extension? What does it contain?
3. Does the server send a **Certificate** message in both connections, or only in the first one?
4. Count the number of handshake packets in each connection. Is the resumed handshake shorter?
5. After the first handshake, find the **NewSessionTicket** message sent by the server. This is the token the client reuses.
6. Why is session resumption useful for performance? Think about both latency (round trips) and computation (which expensive operations are skipped).