OpenSSL Foundation

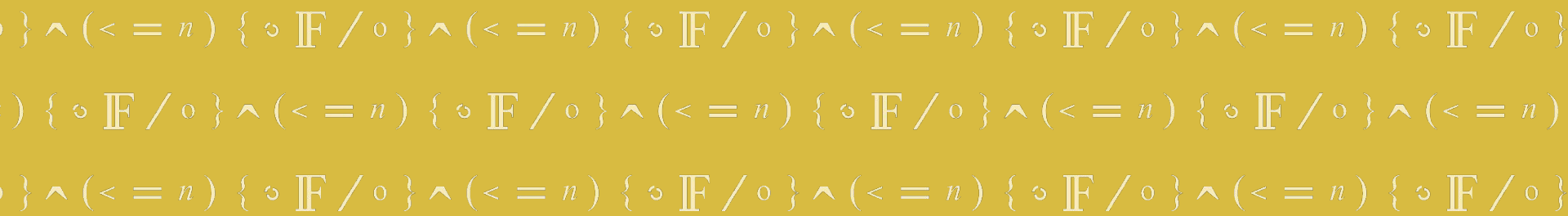
OpenSSL Application Performance Tuning

Tomáš Mráz, Public Support and Security Manager, Director

Agenda

- Building OpenSSL for optimal performance
- Optimizing EVP API usage
- Optimizing libssl API usage
- Choosing the most performant and secure algorithms

Building OpenSSL for optimal performance



Building OpenSSL for optimal performance

- Avoid using `no-asm` – on Windows NASM must be installed
- Avoid using `-d` or `--debug` – that disables optimization
- Add `enable-ec_nistp_64_gcc_128` if performance of P-521 (and P-384 on newer versions) ECDH/ECDSA is important
  - Requires 64 bit little endian platform
- Disable engines (`no-engines`) if not needed
- Disable DH (`no-dh`) or DSA (`no-dsa`) if not needed

Building OpenSSL for optimal performance

- Disabling various other unneeded features is possible
  - It will not affect speed mostly
  - It can lower the footprint of the library in memory
- By default `-O3` is used
  - Try adding `-O2` or `-Os` – these might provide better performance for your concrete use case and platform

Building OpenSSL for optimal performance

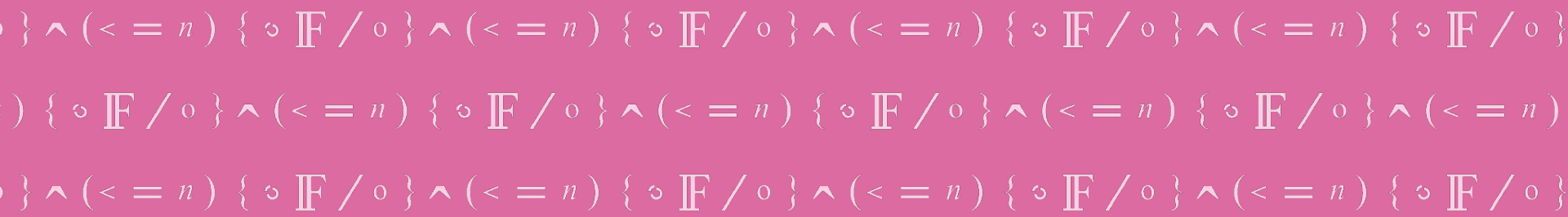
- Last but not least
  - Use the latest 3.x minor version as the performance is improved significantly from 3.0
  - Validated 3.0.8/3.0.9/3.1.2 FIPS providers can be used with the current library providing most of the performance benefits

Building OpenSSL for optimal performance

Minimal TLS implementation with reasonable compatibility and performance

	`./Configure enable-ec_nistp_64_gcc_128`	`./Configure enable-ec_nistp_64_gcc_128 no-argon2 no-aria no-async no-bf no-blake2 no-camellia no-cast no-cmp no-cms no-comp no-deprecated no-des no-dgram no-dh no-dsa no-ec2m no-engine no-gost no-http no-idea no-legacy no-md4 no-mdc2 no-multiblock no-nextprotoneg no-ocb no-ocsp no-quic no-rc2 no-rc4 no-rmd160 no-scrypt no-seed no-siphash no-siv no-sm2 no-sm3 no-sm4 no-srp no-srtp no-ts no-whirlpool -Os`
libcrypto size	6406968 bytes	4270496 bytes (smaller by 33%)
libssl size	1212920 bytes	701304 bytes (smaller by 42%)

Optimizing EVP API usage



Optimizing EVP API usage

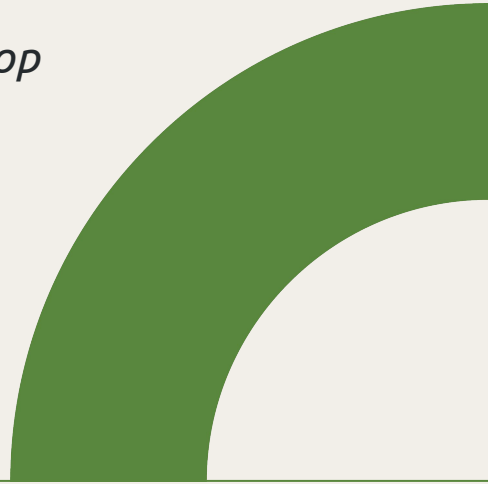
- With support for providers there is much greater flexibility and larger feature set
  - This comes at some costs
- We tried to maximize compatibility with old application code
  - Lazy initialization
    - More locking needed – affects multithreaded performance
  - Implicit algorithm fetching
    - Fetch is not a *free* operation, repeated identical fetches should be avoided
  - Support for engines and low-level *methods* kept
    - Mostly raises the library footprint, does not affect the performance directly

Let’s try to digest 64 bytes with SHA-256 repeatedly

```
for (i = 0; i < num_calls; i++) {
    if ((mctx = EVP_MD_CTX_new()) == NULL)
        goto err;
    if (EVP_DigestInit(mctx, EVP_sha256()) <= 0)
        goto err;
    if (EVP_DigestUpdate(mctx, buf, sizeof(buf)) <= 0)
        goto err;
    if (EVP_DigestFinal(mctx, digest, &dlen) <= 0)
        goto err;
    EVP_MD_CTX_free(mctx);
}
```

Optimizing EVP API usage

- This uses implicit fetch
  - `EVP_sha256()` is just a placeholder object containing the name of the algorithm
  - 0.39µs per one digest operation
- *All numbers were measured on my AMD Ryzen based laptop*
- *With OpenSSL 3.3.1 version*



What if we use explicit digest fetch

```
EVP_MD *md = EVP_MD_fetch(NULL, "SHA-256", NULL);
for (i = 0; i < num_calls; i++) {
    if ((mctx = EVP_MD_CTX_new()) == NULL)
        goto err;
    if (EVP_DigestInit(mctx, md) <= 0)
        goto err;
    if (EVP_DigestUpdate(mctx, buf, sizeof(buf)) <= 0)
        goto err;
    if (EVP_DigestFinal(mctx, digest, &dlen) <= 0)
        goto err;
    EVP_MD_CTX_free(mctx);
}
EVP_MD_free(md);
```

Optimizing EVP API usage

- 0.24µs per one digest operation, this should be roughly equivalent to 1.1.1
  - 38% shorter time
- But we can optimize a little bit more…

No need to allocate and free the EVP_MD_CTX object inside the loop

```
EVP_MD *md = EVP_MD_fetch(NULL, "SHA-256", NULL);
if ((mctx = EVP_MD_CTX_new()) == NULL)
    goto err;
for (i = 0; i < num_calls; i++) {
    if (EVP_DigestInit(mctx, md) <= 0)
        goto err;
    if (EVP_DigestUpdate(mctx, buf, sizeof(buf)) <= 0)
        goto err;
    if (EVP_DigestFinal(mctx, digest, &dlen) <= 0)
        goto err;
}
EVP_MD_CTX_free(mctx);
EVP_MD_free(md);
```

Optimizing EVP API usage

- 0.2µs per one digest operation
  - 49% shorter time
- Now we can apply this to symmetric encryption

Let’s say we need to encrypt 1024 bytes of plaintext with AES-256-CBC periodically

```
if ((cipher = EVP_CIPHER_fetch(NULL, "AES-256-CBC", NULL)) == NULL)
    goto err;
if ((ctx = EVP_CIPHER_CTX_new()) == NULL)
    goto err;
for (i = 0; i < num_calls; i++) {
    if (!EVP_EncryptInit_ex2(ctx, cipher, key, iv, NULL))
        goto err;
    if (!EVP_EncryptUpdate(ctx, ctbuf, &ctlen, ptbuf, sizeof(ptbuf)))
        goto err;
    if (!EVP_EncryptFinal_ex(ctx, ctbuf + ctlen, &ctlen))
        goto err;
}
EVP_CIPHER_CTX_free(ctx);
EVP_CIPHER_free(cipher);
```

Optimizing EVP API usage

- 1.16µs per one 1024 kB encryption
- But we are wasting computation time on something…

We can reuse the preset key schedule

```
if ((cipher = EVP_CIPHER_fetch(NULL, "AES-256-CBC", NULL)) == NULL)
    goto err;
if ((ctx = EVP_CIPHER_CTX_new()) == NULL)
    goto err;
if (!EVP_EncryptInit_ex2(ctx, cipher, key, NULL, NULL))
    goto err;
for (i = 0; i < num_calls; i++) {
    if (!EVP_EncryptInit_ex2(ctx, NULL, NULL, iv, NULL))
        goto err;
    if (!EVP_EncryptUpdate(ctx, ctbuf, &ctlen, ptbuf, sizeof(ptbuf)))
        goto err;
    if (!EVP_EncryptFinal_ex(ctx, ctbuf + ctlen, &ctlen))
        goto err;
}
EVP_CIPHER_CTX_free(ctx);
EVP_CIPHER_free(cipher);
```

Optimizing EVP API usage

- 0.94µs per one 1024 kB encryption
  - 19% shorter time
- What about composite algorithms? (HMAC, HKDF, etc.)

Periodically create a HMAC-SHA-256 of 200 bytes buffer with the same key

```
if ((mac = EVP_MAC_fetch(NULL, "HMAC", NULL)) == NULL)
    goto err;
if ((mctx = EVP_MAC_CTX_new(mac)) == NULL)
    goto err;
params[0] = OSSL_PARAM_construct_utf8_string(OSSL_MAC_PARAM_DIGEST, (char *)digest_name,
                                             sizeof(digest_name));
params[1] = OSSL_PARAM_construct_end();
for (i = 0; i < num_calls; i++) {
    if (!EVP_MAC_init(mctx, key, sizeof(key), params))
        goto err;
    if (!EVP_MAC_update(mctx, buf, sizeof(buf)))
        goto err;
    if (!EVP_MAC_final(mctx, macbuf, &dlen, dlen))
        goto err;
}
EVP_MAC_ctx_free(mctx);
EVP_MAC_free(mac);
```

Optimizing EVP API usage

- This takes 0.74µs per computation of HMAC-SHA-256 over 200 bytes buffer
- However there are also some inefficiencies
  - The SHA-256 digest algorithm is internally fetched in the HMAC implementation
- Can we do better?

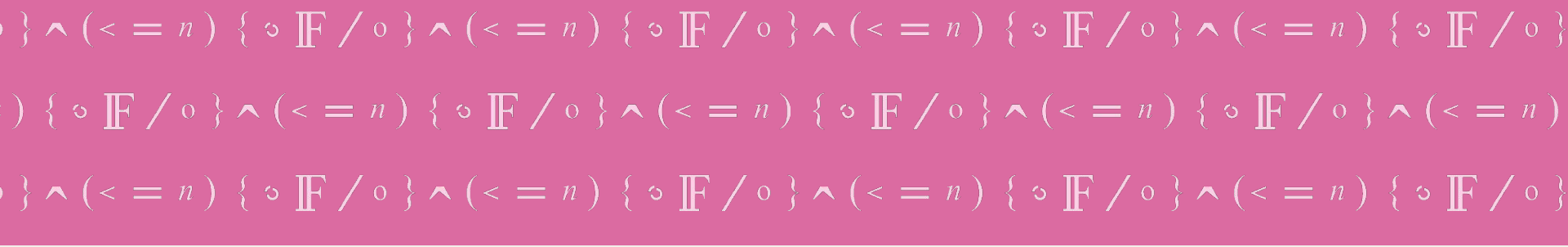
Periodically create a HMAC-SHA-256 of 200 bytes buffer with the same key

```
if ((mac = EVP_MAC_fetch(NULL, "HMAC", NULL)) == NULL)
    goto err;
if ((mctx_tpl = EVP_MAC_CTX_new(mac)) == NULL)
    goto err;
params[0] = OSSL_PARAM_construct_utf8_string(OSSL_MAC_PARAM_DIGEST, (char *)digest_name,
                                             sizeof(digest_name));
params[1] = OSSL_PARAM_construct_end();
if (!EVP_MAC_init(mctx_tpl, key, sizeof(key), params))
    goto err;
for (i = 0; i < num_calls; i++) {
    if (((mctx = EVP_MAC_CTX_dup(mctx_tpl)) == NULL)
        goto err;
    if (!EVP_MAC_update(mctx, buf, sizeof(buf)))
        goto err;
    if (!EVP_MAC_final(mctx, macbuf, &dlen, dlen))
        goto err;
    EVP_MAC_CTX_free(mctx);
    mctx = NULL;
}
...
```

Optimizing EVP API usage

- This takes 0.7µs per computation as we spare both the fetch and the internal initialization of hash context(s)
  - 5% shorter time
- If the key is different we could still use the duplication trick
  - Not showing the code here, left as exercise for you
  - However the context duplication overhead is larger than the internal fetch
  - Requires 0.921µs per computation

Optimizing libssl usage



Optimizing libssl usage

- TLS client
  - Create and preset the SSL_CTX with CA cert store
    - Use the same SSL_CTX with multiple client connections instead of creating it repeatedly for every connection
    - Requires 995µs vs 1100µs to do handshake when the client SSL_CTX is shared
  - Use the cert directory store instead of the cert file store for CA certificates
    - Use `SSL_CTX_load_verify_dir()` instead of `SSL_CTX_load_verify_file()`

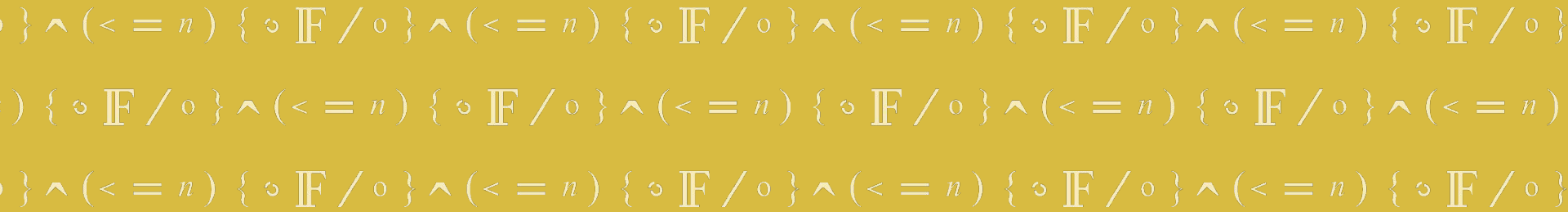
Optimizing libssl usage

- `SSL_SESSION` reuse on the TLS client
  - SSL session caching is enabled by default on the server
  - But not on the client side
  - Even if enabled libssl does not use the cached `SSL_SESSION` entries automatically as only the application knows when the session can be reused
- There is one exception. When an SSL object is reused for a subsequent connection, there is already a SSL_SESSION present in the SSL object.

Optimizing libssl usage

- Avoid using expensive key exchange algorithms
  - Disable RSA and DHE key exchange
    - For TLS-1.2: `SSL_CTX_set_cipher_list(ssl, "DEFAULT:-kRSA:-kDHE");`
    - For TLS-1.3 we need to list the allowed groups
      - For example: `SSL_set1_groups_list(ssl, "X25519:P-256");`
    - Whether X25519 is faster than P-256 depends on the platform
      - On my laptop one X25519 takes 27.5µs, P-256 takes 41.2µs
      - X25519 is not FIPS approved though

How do you choose the most performant algorithms?



Choosing the most performant algorithms

- We started on this topic already
- Never put performance over security
  - Really, so should I always use crypto with at least 256 bit security?
  - That would be silly, so better idea is to say – stick with some minimum security level depending on your requirements (i.e. 128 bit security should be sufficient for all use cases except where 256 bit is mandatory).
  - Use the right algorithm for the job (example: AES-ECB mode encryption is fast, but never the right answer)

Choosing the most performant algorithms

- Which algorithm should I use to symmetrically encrypt some raw data?
- Use an AEAD cipher to protect both confidentiality and integrity of the data.
- To measure use: `openssl speed -evp <algorithm>`

Cipher	Number of bytes processed per second (with 16KiB blocks)
AES-128-GCM	4986 MB/s
AES-128-CCM	1569 MB/s
AES-128-OCB	9031 MB/s
ChaCha20-Poly1305	2315 MB/s

Choosing the most performant algorithms

- The AES-128-OCB is the fastest from these on my hardware
- The results might be very different on different CPU architectures but even different CPUs of the same base architecture
    - For example on CPUs without AES-NI on x86 (or similar instructions on other platforms) the ChaCha20-Poly1305 algorithm will be the fastest one by large margin
- Use `openssl speed` to measure not only the speed of ciphers but also digests, signatures, key exchanges, etc.

Choosing the most performant algorithms

- Which algorithm to choose for certificates?
- Use `openssl speed` to measure RSA vs ECDSA and EdDSA

Algorithm	signatures/s	verifications/s
ECDSA (nistp256)	56539.3	18541.4
RSA-2048	2003.7	69798.3
RSA-3072	660.1	33510.8
EdDSA (Ed25519)	27025.4	10037.5

Choosing the most performant algorithms

- For maximum speed on the server side we want to use ECDSA with nistp256 parameters
- In theory RSA keys (even with 3072 bits to achieve 128 bit security level) would be better for CA certs as the speed of verification is better for RSA
    - However this does not account for larger data transfer size for the intermediate CA cert chain
- EdDSA has some better security properties than ECDSA and could be probably optimized further but currently it is slower than ECDSA with nistp256 on my hardware

Choosing the most performant algorithms

- For maximum compatibility we can have both ECDSA and RSA certificate on the server
- Set the server up in a way to prefer the ECDSA certificate if the client supports it
- For TLS-1.2 we need to order ECDSA ciphers before the RSA

```
SSL_CTX_set_cipher_list(ctx,
"aECDSA:aRSA:-kRSA:-kDHE:-eNULL:-ARIA:-CAMELLIA");
```

Choosing the most performant algorithms

- For TLS-1.3 set the sigalgs
  `SSL_CTX_set1_sigalgs_list(ctx, "ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:ed25519:rsa_pss_pss_sha256:rsa_pss_pss_sha384:rsa_pss_pss_sha512:rsa_pss_rsae_sha256:rsa_pss_rsae_sha384:rsa_pss_rsae_sha512:RSA+SHA256:RSA+SHA384:RSA+SHA512:ECDSA+SHA224:RSA+SHA224");`
- Set server cipher preference: `SSL_CTX_set_options(ctx, SSL_OP_CIPHER_SERVER_PREFERENCE);`
  - This affects both ciphers and sigalgs

Further reading

https://docs.openssl.org/

Thank you! & Questions?