

# Enabling Soft Post-Quantum Cryptography (PQC) on AM62L



Kamlesh Gurudasani, Prashant Shivhare

## Abstract

This application note covers the complete software Post-Quantum Cryptography (PQC) stack for TI AM62Lx platforms running TI SDK 2025.01 (Yocto wrynose). Three approaches are discussed:

1. OpenSSL 3.5 Native — NIST-standardized ML-KEM, ML-DSA, and SLH-DSA built into the base SDK default provider. No additional Yocto integration required. Documented in the [AM62Lx SDK Post-Quantum Cryptography guide](#).
2. liboqs 0.15.0 + oqs-provider 0.11.0 — Extends OpenSSL with hybrid KEM variants (x25519\_mlkem512, p384\_mlkem768), FrodoKEM, and BIKE using meta-oqs. Requires Yocto integration steps Y1–Y2.
3. wolfSSL 5.9.1 — NIST PQC (ML-KEM, ML-DSA) plus stateful hash-based signatures (LMS, XMSS) with `--enable-armasm` NEON acceleration on Cortex-A53. PQC is accessible *only through wolfSSL's native wolfCrypt C API* (`wolfcryptbenchmark/wolfcrypttest`) in this SDK — *not* through `wolfprovider`; it can be used directly by accessing `wolfCrypt` functions from a userspace application. There are two integration paths:
  - Option A (meta-wolfssl): Provides `libwolfssl.so`, `wolfprovider` (hooks wolfSSL's **classical** crypto — RSA, ECC/ECDSA, X25519/X448, Ed25519/Ed448, DH, HMAC, CMAC, HKDF — into the OpenSSL EVP API; *no PQC*), `wolfcryptbenchmark`, and `wolfcrypttest` binaries on target.
  - Option B (meta-oe): Provides `libwolfssl.so` only — native wolfSSL C API, no OpenSSL provider bridge, no benchmark binaries.

All three providers (OpenSSL default, oqs-provider, wolfprovider) are installed together in a single `tisdk-default-image` with unified OpenSSL co-existence configuration. Upstream contributions, provider configuration, and a definitive benchmark comparison on Cortex®-A53 are included.

## Table of Contents

|                                                     |           |
|-----------------------------------------------------|-----------|
| <b>1 Introduction</b>                               | <b>3</b>  |
| 1.1 Platform                                        | 3         |
| <b>2 Yocto Integration (TI SDK Wrynose)</b>         | <b>4</b>  |
| 2.1 Layer Stack                                     | 4         |
| 2.2 OpenSSL 3.5 Native PQC: No Extra Steps Required | 4         |
| 2.3 liboqs / oqs-provider Integration               | 5         |
| 2.4 Recovering From Build Errors                    | 5         |
| 2.5 wolfSSL Integration                             | 6         |
| 2.6 Step B — Build                                  | 9         |
| 2.7 Step C — OpenSSL Provider Configuration         | 9         |
| <b>3 Performance Benchmarks</b>                     | <b>11</b> |
| 3.1 ML-KEM — All implementations                    | 11        |
| 3.2 ML-DSA — All implementations                    | 11        |
| 3.3 FrodoKEM (liboqs only)                          | 12        |
| 3.4 Hybrid KEM — TLS overhead                       | 12        |
| 3.5 armasm Impact on wolfSSL Performance            | 13        |
| <b>4 NEON and HW Acceleration Analysis</b>          | <b>14</b> |
| 4.1 Acceleration Status per Implementation          | 14        |
| 4.2 Recommendation by Algorithm                     | 14        |
| <b>5 Algorithm Availability Summary</b>             | <b>15</b> |

|                           |           |
|---------------------------|-----------|
| <b>6 References</b> ..... | <b>16</b> |
|---------------------------|-----------|

## Trademarks

Cortex® is a registered trademark of Arm Limited.  
All trademarks are the property of their respective owners.

# 1 Introduction

## WARNING

- *OpenSSL 3.5 native (default provider):* Production ready and recommended. NIST-standardized ML-KEM, ML-DSA, and SLH-DSA are fully supported in the OpenSSL 3.5 default provider with no additional restrictions.
- *liboqs / oqs-provider:* The Open Quantum Safe (OQS) project states: "WE DO NOT CURRENTLY RECOMMEND RELYING ON LIBOQS OR OUR APPLICATION INTEGRATIONS IN A PRODUCTION ENVIRONMENT OR TO PROTECT ANY SENSITIVE DATA." See [openquantumsafe.org/about/#limitations](https://openquantumsafe.org/about/#limitations).
- *wolfSSL:* Dual-licensed — GPL v2 (open source) and a commercial license (proprietary products). See [wolfssl.com/license](https://wolfssl.com/license).

OpenSSL 3.5 natively supports NIST PQC standards FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), and FIPS 205 (SLH-DSA). This image extends that baseline with two additional OpenSSL providers:

- *oqs-provider 0.11.0* — hybrid KEM variants (x25519\_mlkem512, p384\_mlkem768) and non-NIST algorithms (FrodoKEM, BIKE) using liboqs 0.15.0
- *wolfprovider 1.1.1* — wolfSSL's *classical* implementations (RSA, ECC/ECDSA, X25519/X448, Ed25519/Ed448, DH, HMAC, CMAC, HKDF) accessible through the OpenSSL EVP API. *Does not include PQC* in this SDK's pinned versions; PQC can be used directly by accessing wolfCrypt functions from a userspace application.

All three providers are active simultaneously through a unified OpenSSL configuration. The image also ships `wolfcryptbenchmark` for direct wolfSSL native benchmarking, and `oqs_speed_kem/oqs_speed_sig` for liboqs direct benchmarking.

## 1.1 Platform

| Item                  | Details                                                     |
|-----------------------|-------------------------------------------------------------|
| Board                 | Texas Instruments AM62L3 Evaluation Module                  |
| Core                  | Cortex-A53 (aarch64), single core                           |
| OS                    | Arago 2025.01 (TI SDK wrynose)                              |
| Kernel                | Linux 6.18.13-ti                                            |
| OpenSSL               | 3.5.5                                                       |
| liboqs                | 0.15.0 (through meta-oqs)                                   |
| oqs-provider          | 0.11.0 (through meta-oqs)                                   |
| wolfSSL               | 5.9.1 (through meta-wolfssl, <code>--enable-armasm</code> ) |
| wolfprovider          | 1.1.1 (through meta-wolfssl)                                |
| CPU extensions active | AES, SHA1, SHA256, SHA512, NEON (OPENSSL_armcap = 0xbd)     |

## Note

### Cortex-A53 hardware crypto capabilities

`OPENSSL_armcap=0xbd` decoded: NEON ✓, AES hardware ✓, PMULL ✓, SHA-1 ✓, SHA-256 ✓, SHA-512 ✓, **SHA-3 ✗** (SHA-3 hardware requires ARMv8.2+; Cortex-A53 is ARMv8.0).

Consequence: ML-KEM (SHAKE-128/256) and ML-DSA (SHAKE-128/256) cannot use SHA-3 hardware. NEON-accelerated software Keccak (liboqs `OQS_USE_ARM_SHA3_INSTRUCTIONS`) and wolfSSL's ARM assembly (`--enable-armasm`) are the available acceleration paths.

## 2 Yocto Integration (TI SDK Wrynose)

The steps in this section assume that the user has a working TI SDK Yocto build environment for AM62Lx already set up. If you have not done this yet, follow the [Building the SDK with Yocto](#) guide first.

The expected directory layout after setup:

```
tisdk/
├── sources/          ← all layers live here
├── build/
│   └── conf/
│       ├── bblayers.conf ← add extra layers here
│       └── local.conf    ← add IMAGE_INSTALL / DISTRO_FEATURES / BBMASK here
```

All path references below use `tisdk/` as the root.

**Build prerequisite:** Before following any steps in this document, you must have sourced the Yocto build environment and completed at least one full image build (Step B — [Section 2.6](#)). These steps are prerequisites and must be done once:

```
cd tisdk/build
. conf/setenv
MACHINE=am62lxx-evm bitbake tisdk-default-image
```

The environment source (`. conf/setenv`) must be re-run in every new terminal session before invoking `bitbake` or running any `bitbake`-based commands.

### 2.1 Layer Stack

```
tisdk/sources/
├── meta-oqs/          ← liboqs + oqs-provider (Steps Y1-Y2)
└── meta-wolfssl/      ← wolfSSL + wolfprovider + wolfcryptbenchmark + --enable-armasm
                        + wolfssl PQC flags bbappend (Option A only)
```

#### Note

wolfSSL has two integration paths:

*Option A (meta-wolfssl): includes `wolfcryptbenchmark` and `wolfcrypttest` binaries. Requires cloning `meta-wolfssl` and 5 BBMASK entries.*

*Option B (meta-oe): uses the `wolfssl` recipe already in `meta-openembedded/meta-networking` — no extra clone, no BBMASK. Library only ( `libwolfssl.so` ), no benchmark binaries.*

`bblayers.conf` additions:

```
BBLAYERS += " \
    ${TOPDIR}/../sources/meta-oqs \
    "${TOPDIR}/../sources/meta-wolfssl \
```

### 2.2 OpenSSL 3.5 Native PQC: No Extra Steps Required

OpenSSL 3.5.5 is included in the base `tisdk-default-image`. The default provider ships with full NIST PQC support; no additional layers, recipes, or `local.conf` entries are needed.

Algorithms available out of the box:

| Category         | Algorithms                                                           |
|------------------|----------------------------------------------------------------------|
| KEM              | ML-KEM-512, ML-KEM-768, ML-KEM-1024                                  |
| Hybrid KEM (TLS) | X25519MLKEM768, X448MLKEM1024, SecP256r1MLKEM768, SecP384r1MLKEM1024 |
| Signature        | ML-DSA-44, ML-DSA-65, ML-DSA-87                                      |
| Hash-based sig   | SLH-DSA (12 variants — SHA-2 and SHAKE, small/fast tradeoffs)        |

For TLS commands (openssl s\_server, openssl s\_client), key generation, signing, certificate creation, and the full TLS handshake walkthrough, see:

- [3.4.6. Post-quantum cryptography — AM62Lx SDK Documentation](#)
- [openssl-s\\_server\(1\)](#), [openssl-s\\_client\(1\)](#), [openssl-genpkey\(1\)](#)

#### Note

What OpenSSL native does not cover:

The base SDK does not include: hybrid KEMs from oqs-provider (x25519\_mlkem512, p384\_mlkem768), non-NIST algorithms (FrodoKEM, BIKE), stateful hash-based signatures (LMS, XMSS), or the wolfSSL NEON-accelerated native API. Steps Y1–Y2 and Option A/B below extend the SDK for these use cases.

## 2.3 liboqs / oqs-provider Integration

This path extends OpenSSL with liboqs's hybrid KEMs, FrodoKEM, and BIKE through oqs-provider. Two steps:

### Step Y1 — Clone meta

```
cd tisdk/sources
git clone https://github.com/embetrix/meta-oqs.git -b wrynose
```

### Step Y2 - Enable OQS in local.conf

```
DISTRO_FEATURES:append = " oqs"
IMAGE_INSTALL:append = " liboqs liboqs-tests oqs-provider"
```

DISTRO\_FEATURES:append = " oqs" is required — it gates the OQS OpenSSL provider config install, oqs-provider RDEPEND, and PQC-aware openssl/ca-certificates bbappends in meta-pqc.

#### WARNING

*liboqs pulls in clang-native/libcxx-native*

*meta-oqs's liboqs\_0.15.0.bb (and liboqs-rust\_0.10.1.bb) set TOOLCHAIN = "clang". This is typically the first thing in a stock TI SDK build that requests the clang toolchain per-recipe — doing so pulls in oe-core's clang-native.bbclass, which depends on **clang-native**, **compiler-rt-native**, and **libcxx-native** (LLVM's own bootstrap of libc++/libc++abi), often for the first time in the whole build.*

If your build's sandboxed *hosttools* environment cannot resolve *-lstdc++* during *libcxx-native*'s own native compiler-ABI check, *do\_configure* for *libcxx-native* fails with *cannot find -lstdc++: No such file or directory*, and **liboqs** never builds. *liboqs* itself has no clang-specific requirement — its SIMD tuning options (OQS\_USE\_AVX2\_INSTRUCTIONS, OQS\_USE\_ARM\_NEON\_INSTRUCTIONS, and so on.) are plain *-m<isa>*-style flags that GCC handles identically. The fix is to force it back to the distro default toolchain using a bbappend in a control layer (for example, *meta-tisdk*):

```
# meta-tisdk/meta-ti-foundational/recipes-crypto/liboqs/liboqs_%.bbappend
TOOLCHAIN = "gcc"
```

## 2.4 Recovering From Build Errors

If any recipe fails during the build, force the recipe to rebuild cleanly:

```
MACHINE=am62lxx-evm bitbake -c cleansstate <recipe>
MACHINE=am62lxx-evm bitbake tisdk-default-image
```

If that does not resolve the issue, also *cleansstate* the recipes it depends on (for example, the library or kernel it links or builds against).

## Example — libwebsockets failure against the current OpenSSL:

```
MACHINE=am62lxx-evm bitbake -c cleansstate libwebsockets openssl
MACHINE=am62lxx-evm bitbake tisdk-default-image
```

Most build failures like this are because of stale or inconsistent build state — a recipe's persistent work/ directory or sstate-cache entry drifting out of sync with its dependencies (for example, after a version bump, a config change, or an interrupted build) — rather than genuine source bugs. cleansstate plus a rebuild is the reliable fix in every case.

## 2.5 wolfSSL Integration

This path extends wolfSSL with PQC and hooks it into OpenSSL. Two integration options are supported, both enabling the same PQC algorithms.

### 2.5.1 Choosing a Path

Two paths are supported. Both enable the same PQC algorithms.

| Requirement                                | Option A: meta-wolfssl | Option B: meta-oe wolfssl |
|--------------------------------------------|------------------------|---------------------------|
| wolfSSL library (libwolfssl.so)            | Yes                    | Yes                       |
| PQC algorithms (ML-KEM, ML-DSA, LMS, XMSS) | Yes                    | Yes                       |
| wolfcryptbenchmark binary on target        | <b>Yes</b>             | No                        |
| wolfcrypttest binary on target             | <b>Yes</b>             | No                        |
| wolfprovider (OpenSSL provider)            | Yes                    | No                        |
| Extra layer clone required                 | Yes (meta-wolfssl)     | No                        |
| BBMASK entries required                    | Yes (5 entries)        | None                      |

Use *Option A* if you need benchmark or test binaries on the board. Use *Option B* if you only need the library for your application.

### 2.5.2 Step A1 — Option A: meta-wolfssl

```
cd tisdk/sources
git clone https://github.com/wolfSSL/meta-wolfssl.git
cd meta-wolfssl
git checkout ae283efad331b392ea6a6ea038030577f35e1b36
cd ../../
```

Add to bblayers.conf:

```
BBLAYERS += " \
    ${TOPDIR}/../sources/meta-wolfssl \
```

Add to local.conf:

```
# Install wolfSSL, benchmark binary, test binary, and wolfprovider (OpenSSL provider)
IMAGE_INSTALL:append = " wolfssl wolfcryptbenchmark wolfcrypttest wolfprovider"
# Mask broken/conflicting wolfssl recipes
BBMASK += "meta-openembedded/meta-networking/recipes-connectivity/wolfssl/"
BBMASK += "meta-wolfssl/*.wolfssl-fips"
BBMASK += "meta-wolfssl/*.wolfssl-linuxkm"
BBMASK += "meta-wolfssl/recipes-examples/wolfengine/wolfenginetest"
BBMASK += "meta-wolfssl/recipes-examples/wolfprovider/wolfproviderenv"
```

## Note

*wolfprovider* is required for Section 2.7

**wolfprovider** must be in **IMAGE\_INSTALL** to enable the unified three-provider OpenSSL configuration in Section 2.7. It is shipped by **meta-wolfssl** and provides **libwolfprov.so**. Without it, the **[wolfprov\_sect]** block in **providers.conf** will fail to load.

All wolfSSL customizations are in **meta-wolfssl/recipes-wolfssl/wolfssl/wolfssl\_%.bbappend** (a version-pinned **wolfssl\_5.9.1.bbappend** works identically — bitbake matches either against the recipe):

```
# meta-wolfssl/recipes-wolfssl/wolfssl/wolfssl_%.bbappend
# Enable ARM assembly (NEON Keccak/SHAKE + hardware AES/SHA-2) on aarch64
EXTRA_OECONF:append:aarch64 = " --enable-armasm"
# Enable NIST PQC algorithms
EXTRA_OECONF:append = " \
    --enable-kyber \
    --enable-dilithium \
    --enable-lms \
    --enable-xmss \
"
```

**--enable-armasm** activates wolfSSL's ARM assembly paths for hardware AES, SHA-256, SHA-512, and **NEON Keccak (SHAKE)** on Cortex-A53 — see [Section 4](#) for full impact analysis.

## WARNING

*armasm + wolfprovider header conflict (aarch64)*

When **--enable-armasm** and **--enable-opensslcoexist** are both active (*wolfprovider* requires **--enable-opensslcoexist**), wolfSSL's installed **wolfssl/wolfcrypt/aes.h** declares an internal symbol **AES\_set\_encrypt\_key** under **#if defined(WOLFSSL\_ARMASM)** with signature **void(const unsigned char\*, word32, unsigned char\*)**. With **OPENSSL\_COEXIST** set in **options.h**, wolfSSL intentionally omits **AES\_BLOCK\_SIZE**, so *wolfprovider's* **internal.h** proceeds to include OpenSSL's **aes.h** — which declares **AES\_set\_encrypt\_key** as **int(const unsigned char\*, const int, AES\_KEY\*)**. Both declarations land in the same translation unit, causing a conflicting-types compiler error.

The fix is in **meta-wolfssl/recipes-wolfssl/wolfssl/wolfssl\_%.bbappend**: a **do\_install:append:aarch64** hook uses **sed** to rename the wolfSSL-internal declaration to **wolfssl\_arm\_AES\_set\_encrypt\_key** in the installed header. This is safe because the wolfSSL **.so** is already compiled with the symbol marked **WOLFSSL\_LOCAL** (GCC hidden visibility), and *wolfprovider* calls wolfSSL AES through **wc\_AesSetKey** — never this internal *armasm* helper directly.

```
# meta-wolfssl/recipes-wolfssl/wolfssl/wolfssl_%.bbappend
do_install:append:aarch64() {
    sed -i 's/^WOLFSSL_LOCAL void AES_set_encrypt_key(/WOLFSSL_LOCAL void
wolfssl_arm_AES_set_encrypt_key(/' \
    ${D}${includedir}/wolfssl/wolfcrypt/aes.h}
```

Because this changes **do\_install** output, **wolfssl** needs its install step rebuilt and **wolfprovider** needs to rebuild against the renamed header:

```
MACHINE=am62lxx-evm bitbake -c cleansstate wolfssl wolfprovider
```

**Note****Benchmark block size (-DBENCH\_EMBEDDED)**

The Yocto wolfcryptbenchmark binary is built with **-DBENCH\_EMBEDDED** (1 KB block size) and **-DUSE\_CERT\_BUFFERS\_2048** (in-RAM certificates). This means:

- **Symmetric cipher throughput** (AES-CBC, AES-GCM, ChaCha20) is measured with 1KB blocks — numbers are higher than a 1MB block run due to L1 cache residency, and are appropriate for embedded TLS use cases (TLS records ≤ 16KB, network packets approximately 1.5KB). The Cortex-A53 L1 cache is 32KB; a 1MB benchmark buffer causes cache thrash and produces throughput numbers that are unrealistic for embedded workloads.
- **PQC ops/sec** (ML-KEM, ML-DSA, LMS, XMSS) are **not affected** — block size is irrelevant to fixed-size keygen/encaps/decaps/sign/verify operations.

The benchmark header shows **(block bytes 1024, min 1.0 sec each)**. This is expected.

**Known flaky mechanism:** the **-DBENCH\_EMBEDDED/-DUSE\_CERT\_BUFFERS\_2048** defines (and the analogous **wolfcrypttest** defines: **-DNO\_WRITE\_TEMP\_FILES -DUSE\_CERT\_BUFFERS\_2048 -DUSE\_CERT\_BUFFERS\_256 -DWOLFSSL\_RSA\_KEY\_CHECK**) are appended to **TARGET\_CFLAGS** through **meta-wolfssl**'s conditionally-require'd **.inc** files, and this mechanism has been observed to silently not reach the compiler on some builds and work correctly on others, with no recipe change in between. If **wolfcrypttest**'s **do\_package\_qa** fails with a **buildpaths** error (an absolute build-host path embedded in the installed binary — not just cosmetic, since the same missing defines mean the binary would also try to open nonexistent build-host paths for some sub-tests on target), re-run **bitbake -c cleansstate wolfssl** and check **grep USE\_CERT\_BUFFERS\_2048 <wolfssl work dir>/temp/log.do\_compile** — if it's absent, the defines didn't propagate that time and a retry (or forcing the same defines via a **bbappend** in your own control layer instead of relying on this nested **.inc** mechanism) is the reliable fix.

**2.5.3 Step A2 — Option B: meta-oe wolfssl (Library Only)**

meta-openembedded/meta-networking is already in the TI SDK **bblayers.conf** — no extra clone is needed. **No BBMASK entries are required.**

**Limitation:** Only **libwolfssl.so** is installed. There are no **wolfcryptbenchmark** or **wolfcrypttest** binaries. The meta-oe recipe ships **wolfssl 5.9.0** (not 5.9.1 as in meta-wolfssl); the *Platform* table in Section 1.1 reflects the Option A configuration.

Add to **local.conf**:

```
# wolfSSL library only (no benchmark/test binaries)
IMAGE_INSTALL:append = " wolfssl"
```

Create the PQC **bbappend** **directly in meta-oe** (this modifies the meta-oe wolfssl recipe to add PQC support):

```
# meta-openembedded/meta-networking/recipes-connectivity/wolfssl/wolfssl_%.bbappend
# Enable ARM assembly (NEON Keccak/SHAKE + hardware AES/SHA-2) on aarch64
EXTRA_OECONF:append:aarch64 = " --enable-armasm"
# Enable NIST PQC algorithms
EXTRA_OECONF:append = " \
    --enable-kyber \
    --enable-dilithium \
    --enable-lms \
    --enable-xmss \
"
```

### WARNING

*Modifying meta-oe directly* This approach edits the meta-oe layer directly. While functional for private SDK work, be aware that:

- Changes can be lost if meta-oe is updated or refreshed
- This is not the recommended Yocto practice for production workflows
- For shared or persistent work, consider using Option A (meta-wolfssl) or creating a dedicated meta-pqc layer instead

If you prefer not to modify meta-oe directly, see Option A above or create a bbappend in your own control layer (for example, meta-tisdk, meta-ti, or meta-custom) that comes after meta-oe in bblayers.conf.

## 2.6 Step B — Build

```
cd tisdk/build
. conf/setenv
MACHINE=am62lxx-evm bitbake tisdk-default-image
```

### Note

Known issue: interrupted builds can leave a recipe's own source checkout dirty

*If a **bitbake** run is interrupted (Ctrl-C) mid-task, some recipes can be left with a half-configured source checkout on the next run, independent of anything in this app note. Symptom seen with **kernel-selftest**: **do\_configure** fails with **The source tree is not clean, please run 'make ARCH=arm64 mrproper'** in **kernel-selftest**'s own unpacked source directory (not the shared kernel build tree — fixing it does not require rebuilding the kernel). Fix is scoped to the affected recipe only:*

*MACHINE = am62lxx-evm bitbake -c cleansstate kernel-selftest*

## 2.7 Step C — OpenSSL Provider Configuration

The three providers are configured in /etc/ssl/openssl.cnf.d/providers.conf. The default image ships wolfprovider.conf and wolfprovider-fips.conf which conflict with openssl-oqs.cnf (duplicate [openssl\_init] and [provider\_sect] section names cause recursive section reference errors). Replace them with a unified config:

```
# On the target board:
mv /etc/ssl/openssl.cnf.d/wolfprovider.conf /tmp/
mv /etc/ssl/openssl.cnf.d/wolfprovider-fips.conf /tmp/

cat > /etc/ssl/openssl.cnf.d/providers.conf << 'EOF'
[openssl_init]
providers = provider_sect
ssl_conf = ssl_module

[provider_sect]
default = default_sect
oqsprovider = oqs_sect
wolfprovider = wolfprov_sect

[default_sect]
activate = 1

[oqs_sect]
module = /usr/lib/openssl-modules/oqsprovider.so
activate = 1

[wolfprov_sect]
module = /usr/lib/openssl-modules/libwolfprov.so

[ssl_module]
system_default = tls_system_default

[tls_system_default]
```

```
Groups = X25519MLKEM768:m1kem768:x25519:prime256v1:x448:secp521r1:secp384r1
EOF
```

### Note

Root cause of the conflict:

*When `/etc/ssl/openssl.cnf` uses `.include` directives, OpenSSL's config parser does **not** restore section context after processing an included file. If `wolfprovider-fips.conf` ends with `[algorithm_sect]`, the parser remains in that section when the next `.include` is processed, causing `openssl_conf = openssl_init` from the following file to be parsed as a key inside `[algorithm_sect]` — triggering `alg_module_init: unknown option`. The fix is unique section names in a single unified file.*

Verify all three providers load:

```
openssl list -providers
```

Expected output:

```
Providers:
  default
    name: OpenSSL Default Provider
    version: 3.5.5
    status: active
  oqsprovider
    name: OpenSSL OQS Provider
    version: 0.11.0
    status: active
  wolfprovider
    name: wolfSSL Provider
    version: 1.1.1
    status: active
```

## 3 Performance Benchmarks

### Note

Prerequisite packages for reproducing benchmarks:

*The commands in this section require the following packages in IMAGE\_INSTALL (see Section 2.3.2 and 2.5.2):*

- *liboqs-tests* → installs *oqs\_speed\_kem*, *oqs\_speed\_sig* to */usr/bin/*
- *wolfcryptbenchmark* → installs */usr/bin/wolfcryptbenchmark* (through *meta-wolfssl* *bbappend*; triggers *--enable-crypttests* in the *wolfssl* build)

*Without these packages, the binaries are not present on the rootfs. Installing the *liboqs* library and *oqsprovider*.so alone is not sufficient for running speed benchmarks.*

All benchmarks run on AM62L3 EVM, Cortex-A53 at approximately 1.4GHz, Arago 2025.01. wolfSSL numbers use *--enable-armasm* (NEON Keccak + hardware AES/SHA-2 assembly active).

All benchmarks run on AM62L3 EVM, Cortex-A53 at approximately 1.4GHz, Arago 2025.01. wolfSSL numbers use **--enable-armasm** (NEON Keccak + hardware AES/SHA-2 assembly active).

### 3.1 ML-KEM — All implementations

```
# OpenSSL native (through EVP provider dispatch)
openssl speed -seconds 3 -kem-algorithms ML-KEM-512 ML-KEM-768 ML-KEM-1024 x25519MLKEM768
# liboqs direct (bypasses OpenSSL EVP)
oqs_speed_kem -d 3 ML-KEM-512
oqs_speed_kem -d 3 ML-KEM-768
oqs_speed_kem -d 3 ML-KEM-1024

# wolfSSL native (--enable-armasm build)
wolfcryptbenchmark -ml-kem-512 -ml-kem-768 -ml-kem-1024
```

| Algorithm   | OpenSSL<br>native<br>keygen/s | liboqs<br>(NEON)<br>keygen/s | wolfSSL<br>keygen/s<br>(+armasm) | OpenSSL<br>encaps/s | liboqs<br>encaps/s | wolfSSL<br>encaps/s<br>(+armasm) | OpenSSL<br>decaps/s | liboqs<br>decaps/s | wolfSSL<br>decaps/s<br>(+armasm) |
|-------------|-------------------------------|------------------------------|----------------------------------|---------------------|--------------------|----------------------------------|---------------------|--------------------|----------------------------------|
| ML-KEM-512  | 4,906                         | 12,829                       | <b>18,897</b>                    | 5,743               | 11,026             | <b>18,355</b>                    | 3,783               | 10,007             | <b>12,876</b>                    |
| ML-KEM-768  | 3,130                         | 7,886                        | <b>11,881</b>                    | 3,963               | 7,299              | <b>11,727</b>                    | 2,668               | 6,482              | <b>8,521</b>                     |
| ML-KEM-1024 | 2,145                         | 5,394                        | <b>7,820</b>                     | 2,882               | 4,963              | <b>6,423</b>                     | 1,994               | 4,405              | <b>5,572</b>                     |

### Note

Why wolfSSL (+armasm) beats liboqs NEON on ML-KEM:

*With **--enable-armasm**, wolfSSL enables aarch64 assembly for **all** crypto primitives — including SHAKE/Keccak (**sha3\_asm.S**). ML-KEM performance is dominated by SHAKE-128/256 (Keccak), so the NEON Keccak assembly gives wolfSSL a **60–100% improvement** over its pure-C baseline and surpasses liboqs by **+47% keygen** and **+67% encaps** on ML-KEM-512.*

*The liboqs NEON NTT path optimises the polynomial arithmetic, but wolfSSL's NEON Keccak speed advantage more than compensates. Without **--enable-armasm**, wolfSSL ML-KEM runs pure-C Keccak and trails liboqs. See [Section 3.5](#) for the before and after comparison.*

### 3.2 ML-DSA — All implementations

```
openssl speed -seconds 3 -signature-algorithms ML-DSA-44 ML-DSA-65 ML-DSA-87
oqs_speed_sig -d 3 ML-DSA-44
oqs_speed_sig -d 3 ML-DSA-65
oqs_speed_sig -d 3 ML-DSA-87
wolfcryptbenchmark -ml-dsa-44 -ml-dsa-65 -ml-dsa-87
```

| Algorithm | OpenSSL<br>keygen/s | liboqs<br>keygen/s | wolfSSL<br>keygen/s<br>(+armasm) | OpenSSL<br>sign/s | liboqs<br>sign/s | wolfSSL<br>sign/s<br>(+armasm) | OpenSSL<br>verify/s | liboqs<br>verify/s | wolfSSL<br>verify/s<br>(+armasm) |
|-----------|---------------------|--------------------|----------------------------------|-------------------|------------------|--------------------------------|---------------------|--------------------|----------------------------------|
| ML-DSA-44 | 1,306               | 1,715              | 3,606                            | 208               | 303              | 834                            | 1,151               | 1,460              | 3,061                            |
| ML-DSA-65 | 805                 | 1,041              | 2,044                            | 121               | 196              | 532                            | 745                 | 953                | 1,937                            |
| ML-DSA-87 | 524                 | 676                | 1,275                            | 106               | 164              | 438                            | 478                 | 590                | 1,171                            |

### Note

Why wolfSSL beats liboqs NEON on ML-DSA:

*liboqs 0.15.0 has **no dedicated aarch64 NTT for ML-DSA** — only an AVX2 path for x86 exists ( `OQS_ENABLE_SIG_m1_dsa_65_avx2` is disabled; no `_aarch64` or `_neon` variant). The NEON benefit in liboqs ML-DSA comes only from NEON-accelerated SHAKE ( `OQS_USE_ARM_SHA3_INSTRUCTIONS` ), not the NTT polynomial arithmetic. wolfSSL's C NTT for Dilithium/ML-DSA is more optimized, beating liboqs by approximately 3× on signing and approximately 2× on keygen/verify.*

## 3.3 FrodoKEM (liboqs only)

```
oqs_speed_kem -d 3 FrodoKEM-640-AES
oqs_speed_kem -d 3 FrodoKEM-976-AES
oqs_speed_kem -d 3 FrodoKEM-1344-AES
```

| Algorithm         | keygen/s | encaps/s | decaps/s | Notes               |
|-------------------|----------|----------|----------|---------------------|
| FrodoKEM-640-AES  | 310      | 270      | 278      | Hardware AES active |
| FrodoKEM-976-AES  | 145      | 120      | 126      | Hardware AES active |
| FrodoKEM-1344-AES | 81       | 67       | 67       | Hardware AES active |

FrodoKEM is approximately 25 to 40 times slower than ML-KEM due to large matrix operations, but benefits directly from hardware AES (`OQS_USE_ARM_AES_INSTRUCTIONS=ON`).

## 3.4 Hybrid KEM — TLS overhead

```
# ML-KEM-768 and X25519MLKEM768 (OpenSSL native, all providers loaded)
openssl speed -seconds 3 -kem-algorithms ML-KEM-768 X25519MLKEM768

# x25519_mlkem512 (oqsprovider, all providers loaded)
openssl speed -seconds 3 -kem-algorithms x25519_mlkem512
# p384_mlkem768 - must exclude wolfprovider (see note below)
OPENSSL_CONF=/dev/null openssl speed -seconds 3 \
  -provider default -provider-path /usr/lib/openssl-modules -provider oqsprovider \
  -kem-algorithms p384_mlkem768
```

### WARNING

*p384\_mlkem768 segfaults when wolfprovider is co-loaded*

*Running `openssl speed ... p384_mlkem768` with the unified `providers.conf` (all three providers active) causes a segfault in oqs-provider 0.11.0. Root cause: interaction between wolfprovider and oqsprovider's P-384 key exchange path. The workaround is to bypass `providers.conf` with `OPENSSL_CONF=/dev/null` and load only `default + oqsprovider` explicitly, as shown above. `x25519_mlkem512` is not affected and runs correctly with all three providers loaded.*

| Algorithm       | Provider          | keygen/s                | encaps/s                | decaps/s                | Bottleneck  |
|-----------------|-------------------|-------------------------|-------------------------|-------------------------|-------------|
| ML-KEM-768      | default (OpenSSL) | 3,130                   | 3,963                   | 2,668                   | Pure PQC    |
| X25519MLKEM768  | default (OpenSSL) | 1,653                   | 1,036                   | 1,211                   | X25519 ECDH |
| x25519_mlkem512 | oqsprovider       | Approximate<br>ly 2,600 | Approximate<br>ly 1,185 | Approximate<br>ly 1,200 | X25519 ECDH |

| Algorithm     | Provider    | keygen/s         | encaps/s         | decaps/s          | Bottleneck                      |
|---------------|-------------|------------------|------------------|-------------------|---------------------------------|
| p384_mlkem768 | oqsprovider | Approximately 21 | Approximately 63 | Approximately 108 | P-384 scalar mult (47ms keygen) |

The PQC component is negligible in hybrid variants — classical ECDH dominates entirely.

### 3.5 armasm Impact on wolfSSL Performance

--enable-armasm on aarch64 enables wolfSSL assembly for: hardware AES, hardware SHA-256/SHA-512, and **NEON Keccak/SHAKE** (sha3\_asm.s). The table below shows the performance delta for PQC algorithms.

| Algorithm   | Operation | Without armasm (ops/s) | With armasm (ops/s) | Improvement  |
|-------------|-----------|------------------------|---------------------|--------------|
| ML-KEM-512  | keygen    | 11,896                 | 18,897              | <b>+59%</b>  |
| ML-KEM-512  | encaps    | 10,316                 | 18,355              | <b>+78%</b>  |
| ML-KEM-512  | decaps    | 7,258                  | 12,876              | <b>+77%</b>  |
| ML-KEM-768  | keygen    | 6,145                  | 11,881              | <b>+93%</b>  |
| ML-KEM-768  | encaps    | 6,135                  | 11,727              | <b>+91%</b>  |
| ML-KEM-768  | decaps    | 4,580                  | 8,521               | <b>+86%</b>  |
| ML-KEM-1024 | keygen    | 3,823                  | 7,820               | <b>+105%</b> |
| ML-KEM-1024 | encaps    | 3,884                  | 6,423               | <b>+65%</b>  |
| ML-KEM-1024 | decaps    | 3,061                  | 5,572               | <b>+82%</b>  |
| ML-DSA-44   | keygen    | 3,080                  | 3,606               | +17%         |
| ML-DSA-44   | sign      | 918                    | 834                 | -9% †        |
| ML-DSA-44   | verify    | 2,938                  | 3,061               | +4%          |
| ML-DSA-65   | keygen    | 1,765                  | 2,044               | +16%         |
| ML-DSA-65   | sign      | 558                    | 532                 | -5% †        |
| ML-DSA-87   | keygen    | 1,173                  | 1,275               | +9%          |
| ML-DSA-87   | sign      | 401                    | 438                 | +9%          |
| ML-DSA-87   | verify    | 1,096                  | 1,171               | +7%          |

† ML-DSA signing uses rejection sampling — iteration count is stochastic and causes benchmark variance of ±10% to ±15%. The observed sign regression for ML-DSA-44/65 is within noise.

#### Note

Why ML-KEM benefits most from armasm:

*ML-KEM uses SHAKE-128/256 (Keccak) as its primary primitive: key generation is approximately 60% Keccak, encapsulation is approximately 70% Keccak. wolfSSL's sha3\_asm.s provides a NEON-optimized Keccak-f[1600] permutation, giving 60% to 100% speedup on all ML-KEM operations. ML-DSA uses SHAKE more sparingly (for hashing, not for the dominant NTT polynomial arithmetic), so the SHAKE speedup translates to a smaller overall improvement.*

## 4 NEON and HW Acceleration Analysis

### 4.1 Acceleration Status per Implementation

| Implementation            | ML-KEM NTT         | ML-DSA NTT               | SHAKE/Keccak             | AES      | SHA-2    |
|---------------------------|--------------------|--------------------------|--------------------------|----------|----------|
| OpenSSL 3.5 default       | Pure C             | Pure C                   | NEON keccak1600 asm      | HW       | HW       |
| liboqs 0.15.0 (Yocto)     | aarch64 NEON NTT ✓ | Pure C (no aarch64 path) | NEON Keccak (XKCP) ✓     | HW ✓     | HW ✓     |
| wolfSSL 5.9.1 (no armasm) | Optimized C        | Optimized C              | Pure C                   | Pure C   | Pure C   |
| wolfSSL 5.9.1 (+armasm)   | Optimized C        | Optimized C              | <b>NEON Keccak asm ✓</b> | HW asm ✓ | HW asm ✓ |

Key observations:

- wolfSSL --enable-armasm enables **NEON Keccak (sha3\_asm.S)** in addition to hardware AES/SHA-2 — this is the primary driver of the 60% to 100% ML-KEM speedup
- liboqs has a **dedicated aarch64 NEON NTT for ML-KEM** — but wolfSSL's NEON Keccak advantage on SHAKE outweighs it, giving wolfSSL+armasm the overall lead on ML-KEM
- liboqs has **no aarch64 NTT for ML-DSA** (AVX2 only in liboqs 0.15.0) — wolfSSL optimized C wins on ML-DSA regardless of armasm
- Neither liboqs nor wolfSSL can use SHA-3 hardware (Cortex-A53 lacks it) — NEON Keccak is the best available for SHAKE

### 4.2 Recommendation by Algorithm

| Use case                           | Recommended path               | Reason                                                            |
|------------------------------------|--------------------------------|-------------------------------------------------------------------|
| ML-KEM key exchange (TLS)          | wolfSSL native API (+armasm)   | Fastest: NEON Keccak asm — 47% to 67% faster than liboqs NEON NTT |
| ML-DSA signatures                  | wolfSSL native API             | Fastest: 3× faster signing than liboqs                            |
| Hybrid KEM for TLS (X25519+ML-KEM) | OpenSSL native X25519MLKEM768  | Minimal overhead, no extra provider                               |
| FrodoKEM                           | liboqs (through oqs-provider)  | Only available implementation                                     |
| LMS/XMSS stateful signatures       | wolfSSL/liboqs native API      | Full implementation (keygen+sign+verify)                          |
| Standalone PQC app (no OpenSSL)    | wolfSSL native C API (+armasm) | Best overall PQC performance, no EVP overhead                     |

## 5 Algorithm Availability Summary

| Algorithm                      | OpenSSL 3.5 native | oqs-provider (liboqs)   | liboqs native | wolfprovider | wolfSSL native |
|--------------------------------|--------------------|-------------------------|---------------|--------------|----------------|
| ML-KEM-512/768/1024            | ✓                  | Hybrid variants only†   | ✓             | — ‡          | ✓              |
| ML-DSA-44/65/87                | ✓                  | OID conflict — blocked† | ✓             | — ‡          | ✓              |
| SLH-DSA (all 12)               | ✓                  | —                       | ✓             | —            | —              |
| X25519MLKEM768, X448MLKEM1024  | ✓ (TLS hybrid)     | —                       | —             | —            | —              |
| x25519_mlkem512, p384_mlkem768 | —                  | ✓                       | —             | —            | —              |
| FrodoKEM-640/976/1344          | —                  | ✓                       | ✓             | —            | —              |
| BIKE-L1/L3/L5                  | —                  | ✓                       | ✓             | —            | —              |
| LMS/HSS                        | Verify only        | —                       | ✓             | —            | ✓              |
| XMSS/XMSS^MT                   | —                  | —                       | ✓             | —            | ✓              |

† oqs-provider v0.11.0 on OpenSSL 3.5 auto-suppresses standalone ML-KEM and ML-DSA to avoid OID conflicts with native implementations.

‡ Per [wolfProvider's own README](#), PQC support requires **OpenSSL 3.6+** and wolfSSL master/v5.9.2-stable+ with an explicit `--enable-pqc` build flag.

This SDK pins **OpenSSL 3.5.5**, so wolfprovider's PQC path is unavailable regardless of wolfSSL's own PQC support.

wolfSSL's ML-KEM/ML-DSA remain real and functional, but accessible **only through the native wolfCrypt C API** (`wolfcryptbenchmark/wolfcrypttest`) — not through `wolfprovider` or the `openssl` CLI.

## 6 References

1. Texas Instruments, [AM62Lx SDK — Post-Quantum Cryptography](#)
2. Open Quantum Safe, [liboqs GitHub](#)
3. Open Quantum Safe, [oqs-provider GitHub](#)
4. Embetrix, [meta-oqs GitHub](#)
5. wolfSSL, [wolfSSL GitHub](#)
6. wolfSSL, [meta-wolfssl GitHub](#)
7. wolfSSL, [meta-wolfssl PR #167 — Add Wrynose LTS support](#)
8. Open Quantum Safe, [OQS Limitations Statement](#)
9. National Institute of Standards and Technology, [NIST FIPS 203 — ML-KEM](#)
10. National Institute of Standards and Technology, [NIST FIPS 204 — ML-DSA](#)
11. National Institute of Standards and Technology, [NIST FIPS 205 — SLH-DSA](#)
12. National Institute of Standards and Technology, [NIST SP 800-208 — LMS/XMSS](#)
13. Yocto Project, [Yocto 6.0 wrynose migration guide](#)
14. Texas Instruments, [AM62LX SDK — Building the SDK with Yocto](#)
15. wolfSSL, [wolfProvider README — Post-Quantum \(NIST FIPS 203/204\)](#)

## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#), [TI's General Quality Guidelines](#), or other applicable terms available either on [ti.com](#) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025