

Application Note

在 AM62L 上使能软件后量子加密 (PQC)



Kamlesh Gurudasani, Prashant Shivhare

摘要

本应用手册涵盖运行 TI SDK 2025.01 (Yocto wrynose) 的 TI AM62Lx 平台的完整软件后量子加密 (PQC) 栈。本文讨论三种方法：

1. OpenSSL 3.5 原生 — NIST 标准化的 ML-KEM、ML-DSA 和 SLH-DSA，内置于基础 SDK 默认提供程序中。无需额外的 Yocto 集成。记录在 [AM62Lx SDK 后量子加密指南](#) 中。
2. liboqs 0.15.0 + oqs-provider 0.11.0 — 使用 meta-oqs 通过混合 KEM 变体 (x25519_mlkem512、p384_mlkem768)、FrodoKEM 和 BIKE 扩展 OpenSSL。需要执行 Yocto 集成步骤 Y1 - Y2。
3. wolfSSL 5.9.1 — NIST PQC (ML-KEM、ML-DSA) 以及有状态哈希签名 (LMS、XMSS)，在 Cortex-A53 上通过 --enable-armasm 启用 NEON 加速。在此 SDK 中，PQC 只能通过 wolfSSL 的原生 wolfCrypt C API (wolfcryptbenchmark/wolfcrypttest) 访问 — 不能通过 wolfprovider 访问；可以通过从用户空间应用程序直接调用 wolfCrypt 函数来使用。有两种集成路径：
 - 选项 A (meta-wolfssl)：在目标机上提供 libwolfssl.so、wolfprovider (将 wolfSSL 的经典加密 — RSA、ECC/ECDSA、X25519/X448、Ed25519/Ed448、DH、HMAC、CMAC、HKDF — 接入 OpenSSL EVP API；不含 PQC)、wolfcryptbenchmark 和 wolfcrypttest 二进制文件。
 - 选项 B (meta-oe)：仅提供 libwolfssl.so — 原生 wolfSSL C API，无 OpenSSL 提供程序桥接，无基准测试二进制文件。

三个提供程序 (OpenSSL 默认、oqs-provider、wolfprovider) 一起安装在单个 tisdsk-default-image 中，并采用统一的 OpenSSL 共存配置。本文包含上游贡献、提供程序配置以及 Cortex®-A53 上的权威基准测试比较。

内容

1 简介.....	3
1.1 平台.....	3
2 Yocto 集成 (TI SDK Wrynose).....	4
2.1 层栈.....	4
2.2 OpenSSL 3.5 原生 PQC：无需额外步骤.....	4
2.3 liboqs/oqs-provider 集成.....	5
2.4 从构建错误中恢复.....	5
2.5 wolfSSL 集成.....	6
2.6 步骤 B — 构建.....	9
2.7 步骤 C — OpenSSL 提供程序配置.....	9
3 性能基准测试.....	11
3.1 ML-KEM — 所有实现.....	11
3.2 ML-DSA — 所有实现.....	11
3.3 FrodoKEM (仅 liboqs).....	12
3.4 混合 KEM — TLS 开销.....	12
3.5 armasm 对 wolfSSL 性能的影响.....	13
4 NEON 与硬件加速分析.....	14
4.1 各实现的加速状态.....	14
4.2 按算法给出的建议.....	14
5 算法可用性摘要.....	15
6 参考资料.....	16

商标

Cortex® is a registered trademark of Arm Limited.
所有商标均为其各自所有者的财产。

1 简介

警告

- **OpenSSL 3.5 原生 (默认提供程序)** : 生产就绪且推荐使用。NIST 标准化的 ML-KEM、ML-DSA 和 SLH-DSA 在 OpenSSL 3.5 默认提供程序中受到完全支持，无额外限制。
- **liboqs / oqs-provider** : Open Quantum Safe (OQS) 项目声明：“我们目前不建议在生产环境中依赖 LIBOQS 或我们的应用程序集成，也不建议使用它们保护任何敏感数据。”请参阅 openquantumsafe.org/about/#limitations。
- **wolfSSL** : 双许可 — GPL v2 (开源) 和商业许可 (专有产品)。请参阅 [wolfssl.com/license](https://www.wolfssl.com/license)。

OpenSSL 3.5 原生支持 NIST PQC 标准 FIPS 203 (ML-KEM)、FIPS 204 (ML-DSA) 和 FIPS 205 (SLH-DSA)。此映像通过两个额外的 OpenSSL 提供程序扩展了该基线：

- **oqs-provider 0.11.0** — 使用 liboqs 0.15.0 的混合 KEM 变体 (x25519_mlkem512、p384_mlkem768) 和非 NIST 算法 (FrodoKEM、BIKE)
- **wolfprovider 1.1.1** — wolfSSL 的经典实现 (RSA、ECC/ECDSA、X25519/X448、Ed25519/Ed448、DH、HMAC、CMAC、HKDF)，可通过 OpenSSL EVP API 访问。此 SDK 的固定版本中 **不包含 PQC**；PQC 可以通过从用户空间应用程序直接调用 wolfCrypt 函数来使用。

三个提供程序通过统一的 OpenSSL 配置同时启用。该映像还提供带 **wolfcryptbenchmark** 用于直接进行 wolfSSL 原生基准测试，以及 **oqs_speed_kem/oqs_speed_sig** 用于直接进行 liboqs 基准测试。

1.1 平台

条目	详细信息
开发板	德州仪器 (TI) AM62L3 评估模块
核心	Cortex-A53 (aarch64)，单核
OS	Arago 2025.01 (TI SDK wrynose)
内核	Linux 6.18.13-ti
OpenSSL	3.5.5
liboqs	0.15.0 (通过 meta-oqs)
oqs-provider	0.11.0 (通过 meta-oqs)
wolfSSL	5.9.1 (通过 meta-wolfssl、--enable-armasm)
wolfprovider	1.1.1 (通过 meta-wolfssl)
CPU 扩展已启用	AES、SHA1、SHA256、SHA512、NEON (OPENSSL_armcap = 0xbd)

备注

Cortex-A53 硬件加密能力

OPENSSL_armcap=0xbd 解码 : NEON ✓、AES 硬件 ✓、PMULL ✓、SHA-1 ✓、SHA-256 ✓、SHA-512 ✓、**SHA-3 ✗** (SHA-3 硬件需要 ARMv8.2+；Cortex-A53 属于 ARMv8.0)。

后果 : ML-KEM (SHAKE-128/256) 和 ML-DSA (SHAKE-128/256) 无法使用 SHA-3 硬件。NEON 加速的软件 Keccak (liboqs OQS_USE_ARM_SHA3_INSTRUCTIONS) 和 wolfSSL 的 ARM 汇编 (--enable-armasm) 是可用的加速路径。

2 Yocto 集成 (TI SDK Wrynose)

本节中的步骤假设用户已为 AM62Lx 搭建好可用的 TI SDK Yocto 构建环境。如果尚未完成，请先按照《使用 Yocto 构建 SDK》指南操作。

设置后的预期目录布局：

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

以下所有路径引用均以 `tisdk/` 为根目录。

构建前提条件：在执行本文档中的任何步骤之前，必须已执行 `source` 命令配置 Yocto 构建环境并完成至少一次完整映像构建（步骤 B — 节 2.6）。这些步骤是前提条件，必须执行一次：

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

在调用 `bitbake` 或运行任何基于 `bitbake` 的命令之前，必须在每个新终端会话中重新执行环境 `source` 脚本（`.conf/setenv`）。

2.1 层栈

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

备注

wolfSSL 有两个集成路径：

选项 A (meta-wolfssl)：包含 `wolfcryptbenchmark` 和 `wolfcrypttest` 二进制文件。需要克隆 `meta-wolfssl` 并添加 5 个 `BBMASK` 条目。

选项 B (meta-oe)：使用 `meta-openembedded/meta-networking` 中已有的 `wolfssl` 配方 — 无需额外克隆，无需 `BBMASK`。仅库 (`libwolfssl.so`)，无基准测试二进制文件。

`bblayers.conf` 新增内容：

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

2.2 OpenSSL 3.5 原生 PQC：无需额外步骤

OpenSSL 3.5.5 已包含在基础 `tisdk-default-image` 中。默认提供程序自带完整的 NIST PQC 支持；无需额外的层、配方或 `local.conf` 条目。

开箱即用的算法：

类别	算法
KEM	ML-KEM-512、ML-KEM-768、ML-KEM-1024
混合 KEM (TLS)	X25519MLKEM768、X448MLKEM1024、SecP256r1MLKEM768、SecP384r1MLKEM1024
签名	ML-DSA-44、ML-DSA-65、ML-DSA-87
基于哈希的签名	SLH-DSA (12 种变体 — SHA-2 和 SHAKE，小/快权衡)

有关 TLS 命令 (`openssl s_server`、`openssl s_client`)、密钥生成、签名、证书创建和完整的 TLS 握手演练，请参阅：

- [3.4.6. 后量子加密 — AM62Lx SDK 文档](#)
- [openssl-s_server\(1\)](#)、[openssl-s_client\(1\)](#)、[openssl-genpkey\(1\)](#)

备注

OpenSSL 原生功能不涵盖：

基础 SDK 不包括：来自 `oqs-provider` 的混合 KEM (`x25519_mlkem512`、`p384_mlkem768`)、非 NIST 算法 (`FrodoKEM`、`BIKE`)、有状态哈希签名 (`LMS`、`XMSS`) 或 `wolfSSL` NEON 加速的原生 API。下面的步骤 Y1 - Y2 和选项 A/B 针对这些用例扩展了 SDK。

2.3 liboqs/oqs-provider 集成

该路径通过 `oqs-provider` 使用 `liboqs` 的混合 KEM、`FrodoKEM` 和 `BIKE` 来扩展 `OpenSSL`。两个步骤：

步骤 Y1 — 克隆 meta

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

步骤 Y2 — 在 local.conf 中启用 OQS

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

`DISTRO_FEATURES:append = " oqs"` 是必需的，它控制 `OQS` `OpenSSL` 提供程序配置安装、`oqs-provider` `RDEPEND`，以及 `meta-pqc` 中的 `PQC` 感知型 `openssl/ca-certificates` `bbappend`。

警告

liboqs 会引入 clang-native/libcxx-native

*meta-oqs 的 `liboqs_0.15.0.bb` (和 `liboqs-rust_0.10.1.bb`) 设置 `TOOLCHAIN = "clang"`。这通常是原版 TI SDK 构建中第一个按配方请求 `clang` 工具链的环节 — 这样做会引入 `oe-core` 的 `clang-native.bbclass`，后者依赖 **`clang-native`**、**`compiler-rt-native`** 和 **`libcxx-native`** (`LLVM` 自身的 `libc++/libc++abi` 自举程序)，这往往是整个构建过程中首次引入。*

如果构建的沙盒 `hosttools` 环境在 `libcxx-native` 自身的原生编译器 ABI 检查期间无法解析 `-lstdc++`，则 `libcxx-native` 的 `do_configure` 会失败，提示 `无法找到 -lstdc++：没有该文件或目录`，并且 `liboqs` 永远无法构建。*liboqs* 本身没有 `clang` 特定要求 — 它的 `SIMD` 调优选项 (`OQS_USE_AVX2_INSTRUCTIONS`、`OQS_USE_ARM_NEON_INSTRUCTIONS` 等) 是普通的 `-m<isa>` 风格标志，`GCC` 处理方式完全相同。修复方法是使用控制层 (例如 `meta-tisd`) 中的 `bbappend` 强制其恢复为发行版默认工具链：

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

2.4 从构建错误中恢复

如果任何配方在构建过程中失败，请强制该配方干净地重新构建：

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

如果这样仍不能解决问题，还需对它所依赖的配方执行 `cleansstate` (例如，它链接或构建所依据的库或内核)。

示例 — libwebsockets 在当前 OpenSSL 下发生失败：

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

大多数此类构建失败都是由于过时或不一致的构建状态 — 配方的持久化 `work/` 目录或 `sstate-cache` 条目与其依赖项不同步（例如，在版本升级、配置更改或构建中断之后）— 而不是真正的源代码错误。在任何情况下，`cleansstate` 加上重新构建都是可靠的修复方法。

2.5 wolfSSL 集成

该路径使用 PQC 扩展 wolfSSL，并将其接入 OpenSSL 中。支持两种集成选项，两者均启用相同的 PQC 算法。

2.5.1 选择路径

支持两种路径。两者均启用相同的 PQC 算法。

要求	选项 A : meta-wolfssl	选项 B : meta-oe wolfssl
wolfSSL 库 (libwolfssl.so)	是	是
PQC 算法 (ML-KEM、ML-DSA、LMS、XMSS)	是	是
目标机上的 wolfcryptbenchmark 二进制文件	是	否
目标机上的 wolfcrypttest 二进制文件	是	否
wolfprovider (OpenSSL 提供程序)	是	否
需要额外的克隆层	是 (meta-wolfssl)	否
需要 BBMASK 条目	是 (5 个条目)	无

如果需要在开发板上使用基准测试或测试二进制文件，请使用选项 A。如果应用程序只需要库，请使用选项 B。

2.5.2 步骤 A1 — 选项 A : meta-wolfssl

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

添加到 `bblayers.conf`：

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

添加到 `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"
```

备注

第 2.7 节需要 wolfprovider

wolfprovider 必须包含在 `IMAGE_INSTALL` 中，才能在第 2.7 节中使能统一的三提供程序 OpenSSL 配置。它由 **meta-wolfssl** 提供，并提供 `libwolfprov.so`。如果没有它，`providers.conf` 中的 `[wolfprov_sect]` 块将无法加载。

所有 wolfSSL 自定义均位于 **meta-wolfssl/recipes-wolfssl/wolfssl/wolfssl_%.bbappend** 中 (版本固定的 **wolfssl_5.9.1.bbappend** 效果相同 — bitbake 会将两者均与配方匹配) :

```
# 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 在 Cortex-A53 上激活 wolfSSL 的 ARM 汇编路径, 用于硬件 AES、SHA-256、SHA-512 和 NEON Keccak (SHAKE) — 完整影响分析请参阅节 4。

警告

armasm + wolfprovider 头文件冲突 (aarch64)

当 **--enable-armasm** 和 **--enable-opensslcoexist** 均激活时 (wolfprovider 需要 **--enable-opensslcoexist**), wolfSSL 安装的 **wolfssl/wolfcrypt/aes.h** 在 **#if defined(WOLFSSL_ARMASM)** 下声明了一个内部符号 **AES_set_encrypt_key**, 签名为 **void(const unsigned char*, word32, unsigned char*)**。当 **options.h** 中设置了 **OPENSSL_COEXIST** 时, wolfSSL 会有意省略 **AES_BLOCK_SIZE**, 因此 wolfprovider 的 **internal.h** 会继续包含 OpenSSL 的 **aes.h** — 后者将 **AES_set_encrypt_key** 声明为 **int(const unsigned char*, const int, AES_KEY*)**。这两个声明都落在同一个翻译单元中, 导致类型冲突的编译器错误。

修复方法位于 **meta-wolfssl/recipes-wolfssl/wolfssl/wolfssl_%.bbappend** 中: 一个 **do_install:append:aarch64** 钩子使用 **sed** 将已安装头文件中的 wolfSSL 内部声明重命名为 **wolfssl_arm_AES_set_encrypt_key**。这是安全的, 因为 wolfSSL .so 在编译时已将该符号标记为 **WOLFSSL_LOCAL** (GCC 隐藏可见性), 并且 wolfprovider 通过 **wc_AesSetKey** 调用 wolfSSL AES — 从不直接调用此内部 **armasm** 辅助函数。

```
# 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}
```

因为这改变了 **do_install** 输出, 所以 **wolfssl** 需要重新执行其安装步骤, **wolfprovider** 需要针对重命名后的头文件重新构建:

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

备注

基准测试块大小 (-DBENCH_EMBEDDED)

Yocto wolfcryptbenchmark 二进制文件使用 **-DBENCH_EMBEDDED** (1KB 块大小) 和 **-DUSE_CERT_BUFFERS_2048** (RAM 内证书) 构建。这意味着：

- **对称密码吞吐量** (AES-CBC、AES-GCM、ChaCha20) 以 1KB 块进行测量 — 由于 L1 缓存驻留，数值高于 1MB 块运行，适用于嵌入式 TLS 用例 (TLS 记录 ≤ 16KB、网络数据包约 1.5KB)。Cortex-A53 L1 缓存为 32KB；1MB 基准测试缓冲区会导致缓存抖动，并产生对嵌入式工作负载不现实的吞吐量数值。
- **PQC 运算/秒** (ML-KEM、ML-DSA、LMS、XMSS) **不受影响** — 块大小与固定大小的密钥生成/封装/解封装/签名/验证运算无关。

基准测试头文件显示 (**block bytes 1024, min 1.0 sec each**)。这是预期结果。

已知的不稳定机制：**-DBENCH_EMBEDDED/-DUSE_CERT_BUFFERS_2048** 定义 (以及类似的 **wolfcrypttest** 定义：**-DNO_WRITE_TEMP_FILES -DUSE_CERT_BUFFERS_2048 -DUSE_CERT_BUFFERS_256 -DWOLFSSL_RSA_KEY_CHECK**) 通过 **meta-wolfssl** 的条件 **require** 的 **.inc** 文件追加到 **TARGET_CFLAGS**，并且已观察到该机制在某些构建中会静默地未传递到编译器，而在其他构建中工作正常，其间没有配方更改。如果 **wolfcrypttest** 的 **do_package_qa** 失败并报告 **buildpaths** 错误 (安装的二进制文件中嵌入了绝对构建主机路径 — 这不仅仅是表面问题，因为相同的缺失定义意味着二进制文件在目标机上也会尝试打开不存在的构建主机路径进行某些子测试)，请重新运行 **bitbake -c cleansstate wolfssl** 并检查 **grep USE_CERT_BUFFERS_2048 <wolfssl work dir>/temp/log.do_compile** — 如果不存在，则说明定义这一次未传播，重试 (或通过您自己控制层中的 **bbappend** 强制执行相同的定义，而不依赖此嵌套 **.inc** 机制) 是可靠的修复方法。

2.5.3 步骤 A2 — 选项 B：meta-oe wolfssl (仅库)

meta-openembedded/meta-networking 已包含在 TI SDK bblayers.conf 中 — 无需额外克隆。不需要 **BBMASK** 条目。

限制：仅安装了 **libwolfssl.so**。不包含 **wolfcryptbenchmark** 或 **wolfcrypttest** 二进制文件。**meta-oe** 配方提供 **wolfssl 5.9.0** (而不是 **meta-wolfssl** 中的 5.9.1)；第 1.1 节中的平台表反映的是选项 A 配置。

添加到 **local.conf**：

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

直接在 **meta-oe** 中创建 PQC **bbappend** (这会修改 **meta-oe wolfssl** 配方以添加 PQC 支持)：

```
# 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 \
"
```

警告

直接修改 **meta-oe**，此方法会直接编辑 **meta-oe** 层。虽然可用于私有 SDK 工作，但请注意：

- 如果 **meta-oe** 被更新或刷新，更改可能会丢失
- 这不是生产工作流程推荐的 Yocto 实践
- 对于共享或持久性工作，请考虑使用选项 A (**meta-wolfssl**) 或创建专用的 **meta-pqc** 层

如果不想直接修改 **meta-oe**，请参阅上面的选项 A，或在您自己的控制层（例如 **meta-tisdk**、**meta-ti** 或 **meta-custom**）中创建 **bbappend**，该层在 **bbayers.conf** 中位于 **meta-oe** 之后。

2.6 步骤 B — 构建

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

备注

已知问题：中断的构建可能导致配方自身的源代码检出处于脏状态

如果一次 **bitbake** 运行在任务中途中断 (**Ctrl-C**)，某些配方在下次运行时可能遗留半配置的源代码检出，该现象与本应用手册中的内容无关。在 **kernel-selftest** 中观察到的症状：**do_configure** 失败，提示 **源代码树不干净**，请在 **kernel-selftest** 自己的解包源代码目录中运行 **'make ARCH=arm64 mrproper'**（不是共享的内核构建树 — 修复它不需要重建内核）。修复方法仅限于受影响的配方：

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

2.7 步骤 C — OpenSSL 提供程序配置

三个提供程序在 **/etc/ssl/openssl.cnf.d/providers.conf** 中配置。默认映像附带 **wolfprovider.conf** 和 **wolfprovider-fips.conf**，这两者与 **openssl-oqs.cnf** 冲突（重复的 **[openssl_init]** 和 **[provider_sect]** 段名会导致递归段引用错误）。将它们替换为统一配置：

```
# 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:mlkem768:x25519:prime256v1:x448:secp521r1:secp384r1
EOF
```

备注

冲突的根本原因：

当 `/etc/ssl/openssl.cnf` 使用 `.include` 指令时，OpenSSL 的配置解析器在处理完包含的文件后不会恢复段上下文。如果 `wolfprovider-fips.conf` 以 `[algorithm_sect]` 结尾，则解析器在处理下一个 `.include` 时仍停留在该段中，导致后续文件中的 `openssl_conf = openssl_init` 被解析为 `[algorithm_sect]` 内的一个键 — 从而触发 `alg_module_init: unknown option`。修复方法是在单个统一文件中使用唯一的段名。

验证三个提供程序是否均已加载：

```
openssl list -providers
```

预期输出：

```
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 性能基准测试

备注

重现基准测试所需的前提包：

本节中的命令需要 `IMAGE_INSTALL` 中的以下包 (请参阅第 2.3.2 节和 2.5.2 节)：

- `liboqs-tests` → 将 `oqs_speed_kem`、`oqs_speed_sig` 安装到 `/usr/bin/`
- `wolfcryptbenchmark` → 安装 `/usr/bin/wolfcryptbenchmark` (通过 `meta-wolfssl bbappend`；在 `wolfssl` 构建中触发 `--enable-crypttests`)

如果没有这些包，`rootfs` 上将不会包含二进制文件。仅安装 `liboqs` 库和 `oqsprovider.so` 并不足以运行速度基准测试。

所有基准测试均在 AM62L3 EVM 上运行，该平台搭载频率约为 1.4GHz 的 Cortex-A53，并采用 Arago 2025.01。wolfSSL 数值使用 `--enable-armasm` (NEON Keccak + 硬件 AES/SHA-2 汇编已启用)。

所有基准测试均在 AM62L3 EVM 上运行，该平台搭载频率约为 1.4GHz 的 Cortex-A53，并采用 Arago 2025.01。wolfSSL 数值使用 `--enable-armasm` (NEON Keccak + 硬件 AES/SHA-2 汇编已启用)。

3.1 ML-KEM — 所有实现

```
# 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
```

算法	OpenSSL 原生密钥生 成/秒	liboqs (NEON) 密 钥生成/秒	wolfSSL 密钥 生成/秒 (+armasm)	OpenSSL 封装/秒	liboqs 封 装/秒	wolfSSL 封 装/秒 (+armasm)	OpenSSL 解封装/秒	liboqs 解 封装/秒	wolfSSL 解封 装/秒 (+armasm)
ML- KEM-512	4,906	12,829	18,897	5,743	11,026	18,355	3,783	10,007	12,876
ML- KEM-768	3,130	7,886	11,881	3,963	7,299	11,727	2,668	6,482	8,521
ML- KEM-1024	2,145	5,394	7,820	2,882	4,963	6,423	1,994	4,405	5,572

备注

为什么 wolfSSL (+armasm) 在 ML-KEM 上击败了 liboqs NEON：

使用 `--enable-armasm` 时，wolfSSL 为所有加密原语使能 `aarch64` 汇编 — 包括 `SHAKE/Keccak (sha3_asm.S)`。ML-KEM 性能主要由 `SHAKE-128/256 (Keccak)` 决定，因此 NEON Keccak 汇编使 wolfSSL 相比其纯 C 基线提升 60 - 100%，并在 ML-KEM-512 上在密钥生成方面提升 47%、在封装方面提升 67% 超越 liboqs。

liboqs 的 NEON NTT 路径优化了多项式运算，但 wolfSSL 的 NEON Keccak 速度优势足以弥补这一差距。如果不使用 `--enable-armasm`，wolfSSL ML-KEM 将运行纯 C Keccak，并落后于 liboqs。有关前后对比，请参阅节 3.5。

3.2 ML-DSA — 所有实现

```
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
```

算法	OpenSSL 密钥生成/秒	liboqs 密钥 生成/秒	wolfSSL 密钥 生成/秒 (+armasm)	OpenSSL 签名/秒	liboqs 签名/秒	wolfSSL 签 名/秒 (+armasm)	OpenSSL 验证/秒	liboqs 验 证/秒	wolfSSL 验 证/秒 (+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

备注

为什么 wolfSSL 在 ML-DSA 上优于 liboqs NEON：

liboqs 0.15.0 没有针对 ML-DSA 的专用 aarch64 NTT — 仅存在 x86 的 AVX2 路径 (OQS_ENABLE_SIG_ml_dsa_65_avx2 被禁用；没有 _aarch64 或 _neon 变体)。liboqs ML-DSA 中的 NEON 优势仅来自 NEON 加速的 SHAKE (OQS_USE_ARM_SHA3_INSTRUCTIONS)，而非 NTT 多项式运算。wolfSSL 针对 Dilithium/ML-DSA 的 C NTT 优化程度更高，在签名上比 liboqs 快约 3 倍，在密钥生成/验证上快约 2 倍。

3.3 FrodoKEM (仅 liboqs)

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

算法	密钥生成/秒	封装/秒	解封装/秒	备注
FrodoKEM-640-AES	310	270	278	硬件 AES 已启用
FrodoKEM-976-AES	145	120	126	硬件 AES 已启用
FrodoKEM-1344-AES	81	67	67	硬件 AES 已启用

由于要执行大型矩阵运算，FrodoKEM 比 ML-KEM 慢约 25 至 40 倍，但可直接受益于硬件 AES (OQS_USE_ARM_AES_INSTRUCTIONS=ON)。

3.4 混合 KEM — TLS 开销

```
# 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
```

警告

当 wolfprovider 被同时加载时，p384_mlkem768 会发生段错误

使用统一的 providers.conf (所有三个提供程序均已启用) 运行 openssl speed ... p384_mlkem768 会导致 oqs-provider 0.11.0 发生段错误。根本原因：wolfprovider 与 oqsprovider 的 P-384 密钥交换路径之间的交互。解决方法是使用 OPENSSL_CONF=/dev/null 绕过 providers.conf，并仅显式加载 default + oqsprovider，如上所示。x25519_mlkem512 不受影响，并在加载三个提供程序时正常运行。

算法	提供程序	密钥生成/秒	封装/秒	解封装/秒	瓶颈
ML-KEM-768	默认 (OpenSSL)	3,130	3,963	2,668	纯 PQC

算法	提供程序	密钥生成/秒	封装/秒	解封装/秒	瓶颈
X25519MLKEM768	默认 (OpenSSL)	1,653	1,036	1,211	X25519 ECDH
x25519_mlkem512	oqsprovider	约 2,600	约 1,185	约 1,200	X25519 ECDH
p384_mlkem768	oqsprovider	约 21	约 63	约 108	P-384 标量乘法 (47ms 密钥生成)

在混合变体中，PQC 组件可忽略不计 — 经典 ECDH 完全占主导。

3.5 armasm 对 wolfSSL 性能的影响

--enable-armasm 在 aarch64 上为以下算法使能 wolfSSL 汇编：硬件 AES、硬件 SHA-256/SHA-512 和 **NEON Keccak/SHAKE** (sha3_asm.s)。下表显示了 PQC 算法的性能变化。

算法	操作	未使用 armasm (操作/秒)	使用 armasm (操作/秒)	提升幅度
ML-KEM-512	密钥生成	11,896	18,897	+59%
ML-KEM-512	封装	10,316	18,355	+78%
ML-KEM-512	解封装	7,258	12,876	+77%
ML-KEM-768	密钥生成	6,145	11,881	+93%
ML-KEM-768	封装	6,135	11,727	+91%
ML-KEM-768	解封装	4,580	8,521	+86%
ML-KEM-1024	密钥生成	3,823	7,820	+105%
ML-KEM-1024	封装	3,884	6,423	+65%
ML-KEM-1024	解封装	3,061	5,572	+82%
ML-DSA-44	密钥生成	3,080	3,606	+17%
ML-DSA-44	签名	918	834	-9% †
ML-DSA-44	验证	2,938	3,061	+4%
ML-DSA-65	密钥生成	1,765	2,044	+16%
ML-DSA-65	签名	558	532	-5% †
ML-DSA-87	密钥生成	1,173	1,275	+9%
ML-DSA-87	签名	401	438	+9%
ML-DSA-87	验证	1,096	1,171	+7%

† ML-DSA 签名使用拒绝采样 — 迭代次数是随机的，会导致 $\pm 10\%$ 至 $\pm 15\%$ 的基准测试波动。观察到的 ML-DSA-44/65 签名回退在噪声范围内。

备注

为什么 ML-KEM 从 armasm 获益最大：

ML-KEM 使用 SHAKE-128/256 (Keccak) 作为其主要原语：密钥生成中约 60% 为 Keccak，封装中约 70% 为 Keccak。wolfSSL 的 sha3_asm.s 提供了 NEON 优化的 Keccak-f[1600] 置换，使所有 ML-KEM 操作获得 60% 至 100% 的加速。ML-DSA 对 SHAKE 的使用更少（用于哈希，而非占主导的 NTT 多项式算术），因此 SHAKE 加速带来的整体改进较小。

4 NEON 与硬件加速分析

4.1 各实现的加速状态

实现	ML-KEM NTT	ML-DSA NTT	SHAKE/Keccak	AES	SHA-2
OpenSSL 3.5 默认配置	纯 C	纯 C	NEON keccak1600 汇编	硬件	硬件
liboqs 0.15.0 (Yocto)	aarch64 NEON NTT ✓	纯 C (无 aarch64 路径)	NEON Keccak (XKCP) ✓	硬件 ✓	硬件 ✓
wolfSSL 5.9.1 (无 armasm)	优化型 C	优化型 C	纯 C	纯 C	纯 C
wolfSSL 5.9.1 (+armasm)	优化型 C	优化型 C	NEON Keccak asm ✓	硬件 asm ✓	硬件 asm ✓

重要观察结果：

- wolfSSL --enable-armasm 除硬件 AES/SHA-2 外还使能 **NEON Keccak (sha3_asm.S)** — 这是 60% 至 100% ML-KEM 加速的主要驱动因素
- liboqs 为 **ML-KEM** 提供了专用的 **aarch64 NEON NTT** — 但 wolfSSL 在 SHAKE 上的 NEON Keccak 优势更为显著，使 wolfSSL+armasm 在 ML-KEM 上整体领先
- liboqs 没有针对 **ML-DSA** 的 **aarch64 NTT** (liboqs 0.15.0 中仅有 AVX2) — 无论有无 armasm，wolfSSL 优化型 C 在 ML-DSA 上均更优
- liboqs 和 wolfSSL 都无法使用 SHA-3 硬件 (Cortex-A53 不具备该功能) — NEON Keccak 是 SHAKE 可用的最佳方案

4.2 按算法给出的建议

用例	建议路径	原因
ML-KEM 密钥交换 (TLS)	wolfSSL 原生 API (+armasm)	最快：NEON Keccak 汇编 — 比 liboqs NEON NTT 快 47% 到 67%
ML-DSA 签名	wolfSSL 原生 API	最快：签名比 liboqs 快 3 倍
用于 TLS 的混合 KEM (X25519+ML-KEM)	OpenSSL 原生 X25519MLKEM768	开销最小，无需额外提供程序
FrodoKEM	liboqs (通过 oqs-provider)	唯一可用的实现
LMS/XMSS 有状态签名	wolfSSL/liboqs 原生 API	完整实现 (密钥生成 + 签名 + 验证)
独立 PQC 应用程序 (无 OpenSSL)	wolfSSL 原生 C API (+armasm)	整体 PQC 性能最佳，无 EVP 开销

5 算法可用性摘要

算法	OpenSSL 3.5 原生	oqs-provider (liboqs)	liboqs 原生	wolfprovider	wolfSSL 原生
ML-KEM-512/768/1024	✓	仅混合变体†	✓	— ‡	✓
ML-DSA-44/65/87	✓	OID 冲突 — 已阻止†	✓	— ‡	✓
SLH-DSA (全部 12 种)	✓	—	✓	—	—
X25519MLKEM768、 X448MLKEM1024	✓ (TLS 混合)	—	—	—	—
x25519_mlkem512、p384_mlkem768	—	✓	—	—	—
FrodoKEM-640/976/1344	—	✓	✓	—	—
BIKE-L1/L3/L5	—	✓	✓	—	—
LMS/HSS	仅验证	—	✓	—	✓
XMSS/XMSS^MT	—	—	✓	—	✓

† 在 OpenSSL 3.5 上，oqs-provider v0.11.0 会自动抑制独立的 ML-KEM 和 ML-DSA，以避免与原生实现发生 OID 冲突。

‡ 根据 [wolfProvider 的自述文件](#)，PQC 支持需要 **OpenSSL 3.6+** 和 wolfSSL master/v5.9.2-stable+，并使用显式 `--enable-pqc` 构建标志。

此 SDK 固定使用 **OpenSSL 3.5.5**，因此无论 wolfSSL 自身是否支持 PQC，wolfprovider 的 PQC 路径都不可用。

wolfSSL 的 ML-KEM/ML-DSA 仍然可用且功能正常，但只能通过原生 **wolfCrypt C API** (wolfcryptbenchmark/wolfcrypttest) 访问 — 不能通过 wolfprovider 或 openssl CLI 访问。

6 参考资料

1. 德州仪器 (TI), [AM62Lx SDK — 后量子加密](#)
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 — 添加 Wrynose LTS 支持](#)
8. Open Quantum Safe, [OQS 限制声明](#)
9. 美国国家标准与技术研究院, [NIST FIPS 203 — ML-KEM](#)
10. 美国国家标准与技术研究院, [NIST FIPS 204 — ML-DSA](#)
11. 美国国家标准与技术研究院, [NIST FIPS 205 — SLH-DSA](#)
12. 美国国家标准与技术研究院, [NIST SP 800-208 — LMS/XMSS](#)
13. Yocto Project, [Yocto 6.0 wrynose 迁移指南](#)
14. 德州仪器 (TI), [AM62LX SDK — 使用 Yocto 构建 SDK](#)
15. wolfSSL, [wolfProvider README — 后量子 \(NIST FIPS 203/204\)](#)

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 [TI 销售条款](#)、[TI 通用质量指南](#) 或 [ti.com](#) 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

版权所有 © 2026，德州仪器 (TI) 公司

最后更新日期：2025 年 10 月