

Application Note

AM62L でのソフト ポスト量子暗号 (PQC) の有効化



Kamlesh Gurudasani, Prashant Shivhare

概要

このアプリケーション ノートでは、TI SDK 2025.01 (Yocto wrynose) が動作する TI AM62Lx プラットフォーム用の包括的なソフトウェアであるポスト量子暗号 (PQC) スタックについて説明しています。説明するのは、次の 3 つのアプローチについてです。

1. **OpenSSL 3.5 ネイティブ**: ベース SDK のデフォルト プロバイダに組み込まれている NIST 標準の ML-KEM、ML-DSA、および SLH-DSA。Yocto の追加の統合は不要です。AM62Lx SDK ポスト量子暗号ガイドに記載されています。
2. **liboqs 0.15.0+ oqs-provider 0.11.0** — ハイブリッド KEM バリエント (x25519_mlkem512、p384_mlkem768)、FrodoKEM、および meta-oqs を使用した BIKE で OpenSSL を拡張します。Yocto 統合ステップ Y1 ~ Y2 が必要です。
3. **wolfSSL 5.9.1** — NIST PQC (ML-KEM、ML-DSA) と、Cortex-A53 上の `--enable-armasm` NEON アクセラレーションを使用したステートフル ハッシュベースのシグネチャ (LMS、XMSS)。PQC には、本 SDK では wolfSSL のネイティブ wolfCrypt C API (`wolfcryptbenchmark`/`wolfcrypttest`) からのみアクセスできます。wolfprovider 経由ではなく、ユーザー空間アプリケーションから wolfCrypt 関数にアクセスすることで直接利用できます。次の 2 つの統合パスがあります。
 - オプション A (meta-wolfssl): `libwolfssl.so`、wolfprovider (wolfSSL の古典的な暗号 (RSA、ECC/ECDSA、X25519/X448、Ed25519/Ed448、DH、HMAC、CMAC、HKDF) を PQC なしの OpenSSL EVP API にフック)、wolfcryptbenchmark および wolfcrypttest バイナリをターゲットに提供します。
 - オプション B (meta-oe): ネイティブ wolfSSL C API である `libwolfssl.so` のみを提供し、OpenSSL プロバイダブリッジ、ベンチマーク バイナリは提供しません。

3 つのプロバイダ (OpenSSL デフォルト、oqs-provider、wolfprovider) はすべて、OpenSSL の統一された共存構成を使用した単一の `tisdk-default-image` にまとめてインストールされます。アップストリームへの貢献、プロバイダの構成、Cortex®-A53 に関する最終的なベンチマーク比較も組み込まれています。

目次

1 概要.....	3
1.1 プラットフォーム.....	3
2 Yocto 統合 (TI SDK Wrynose).....	4
2.1 積層.....	4
2.2 OpenSSL 3.5 ネイティブ PQC: 追加のステップは不要.....	4
2.3 liboqs/oqs-provider 統合.....	5
2.4 ビルド エラーからの回復.....	6
2.5 wolfSSL の統合.....	6
2.6 ステップ B — ビルド.....	9
2.7 ステップ C — OpenSSL プロバイダの構成.....	9
3 性能ベンチマーク.....	11
3.1 ML-KEM — すべての実装.....	11
3.2 ML-DSA — すべての実装.....	12
3.3 FrodoKEM (liboqs のみ).....	12
3.4 ハイブリッド KEM — TLS オーバーヘッド.....	12
3.5 wolfSSL の性能に対する armasm の影響.....	13
4 NEON および HW アクセラレーション分析.....	15

4.1 実装ごとのアクセラレーション ステータス.....	15
4.2 アルゴリズムによる推奨.....	15
5 アルゴリズム可用性の概要.....	16
6 参考資料.....	17

商標

Cortex® is a registered trademark of Arm Limited.

すべての商標は、それぞれの所有者に帰属します。

1 概要

警告

- **OpenSSL 3.5** ネイティブ (デフォルト プロバイダ): 量産に対応可能で、推奨されます。NIST 標準の **ML-KEM**、**ML-DSA**、および **SLH-DSA** は、**OpenSSL 3.5** のデフォルト プロバイダで完全にサポートされており、追加の制限はありません。
- **liboqs/oqs-provider**: **Open Quantum Safe (OQS)** プロジェクトでは、次のように規定しています。「現在、本番環境で **LIBOQS** やアプリケーション統合を使用することや、機密データを保護することは推奨していません。」openquantumsafe.org/about/#limitations を参照してください。
- **wolfSSL**: デュアル ライセンス — **GPL v2** (オープン ソース) および商用ライセンス (独自製品)。
[wolfssl.com/license](https://www.wolfssl.com/license) を参照してください。

OpenSSL 3.5 は、NIST PQC 標準 **FIPS 203 (ML-KEM)**、**FIPS 204 (ML-DSA)**、**FIPS 205 (SLH-DSA)** をネイティブでサポートしています。このイメージは、次の 2 つの追加 **OpenSSL** プロバイダを使用してベースラインを拡張します。

- **oqs-provider 0.11.0** — **liboqs 0.15.0** を使用したハイブリッド **KEM** バリエント (**x25519_mlkem512**、**p384_mlkem768**) および非 NIST アルゴリズム (**FrodoKEM**、**BIKE**)
- **wolfprovider 1.1.1** — **wolfSSL** の古典的な実装 (**RSA**、**ECC/ECDSA**、**X25519/X448**、**Ed25519/Ed448**、**DH**、**HMAC**、**CMAC**、**HKDF**) には、**OpenSSL EVP API** からアクセスできます。PQC は本 SDK の固定バージョンには含まれていません。PQC を直接使用するには、ユーザースペース アプリケーションから **wolfCrypt** 関数にアクセスします。

3 つのプロバイダはすべて、**OpenSSL** の統合構成を介して同時にアクティブになります。このイメージには、**wolfSSL** の直接ネイティブ ベンチマーク用の **wolfcryptbenchmark** と、**liboqs** の直接ベンチマーク用の **oqs_speed_kem/oqs_speed_sig** も付属しています。

1.1 プラットフォーム

項目	詳細
ボード	テキサス インストルメンツ AM62L3 評価基板
コア	Cortex-A53 (aarch64) 、シングル コア
OS	Arago 2025.01 (TI SDK wrynose)
カーネル	Linux 6.18.13-ti
OpenSSL	3.5.5
liboqs	0.15.0 (meta-oqs 経由)
oqs-provider	0.11.0 (meta-oqs 経由)
wolfSSL	5.9.1 (meta-wolfssl,--enable-armasm 経由)
wolfprovider	1.1.1 (meta-wolfssl 経由)
CPU 拡張アクティブ	AES 、 SHA1 、 SHA256 、 SHA512 、 NEON (OPENSSL_armcap = 0xbd)

注

Cortex-A53 のハードウェア暗号化機能

OPENSSL_armcap=0xbd のデコード: **NEON**✓、**AES** ハードウェア✓、**PMULL**✓、**SHA-1**✓、**SHA-256**✓、**SHA-512**✓、**SHA-3** ✗ (**SHA-3** ハードウェアには **ARMv8.2+** が必要、**Cortex-A53** は **ARMv8.0**)。

結論: **ML-KEM (SHAKE-128/256)** と **ML-DSA (SHAKE-128/256)** では **SHA-3** ハードウェアを使用できません。NEON 高速化ソフトウェア **Keccak (liboqs OQS_USE_ARM_SHA3_INSTRUCTIONS)** と **wolfSSL** の **ARM アセンブリ (--enable-armasm)** が、利用可能な高速化パスです。

2 Yocto 統合 (TI SDK Wrynose)

このセクションのステップでは、AM62Lx 用に動作する TI SDK Yocto ビルド環境をユーザーがすでに設定していることが想定されています。まだこの作業を行っていない場合は、まず『[Building the SDK with Yocto](#)』ガイドの指示に従ってください。

設定後に必要なディレクトリレイアウト:

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

以下のパス参照はすべて、ルートとして `tisdsk/` を使用します。

ビルドの前提条件: このドキュメントのステップに従う前に、Yocto ビルド環境の設定と、少なくとも 1 つのフルイメージのビルドが完了している必要があります (ステップ B — [セクション 2.6](#))。これらのステップは前提条件であり、1 回実行されていなければなりません。

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

新しいターミナル セッションごとに環境ソース (`conf/setenv`) を再実行してから、`bitbake` を起動するか、`bitbake` ベースのコマンドを実行する必要があります。

2.1 積層

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

注

wolfSSL には次の 2 つの統合パスがあります。

オプション A (`meta-wolfssl`): `wolfcryptbenchmark` と `wolfcrypttest` のバイナリが含まれます。`meta-wolfssl` と 5 つの `BBMASK` エントリのクローニングが必要です。

オプション B (`meta-oe`): `meta-openembedded/meta-networking` に既にある `wolfssl` レシピを使用します。追加クローンも `BBMASK` も使用しません。ライブラリのみで (`libwolfssl.so`)、ベンチマーク バイナリはありません。

`bblayers.conf` `bblayers.conf` の追加:

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

2.2 OpenSSL 3.5 ネイティブ PQC: 追加のステップは不要

OpenSSL 3.5.5 はベース `tisdsk-default-image` に組み込まれています。デフォルトのプロバイダには、NIST PQC のフルサポートが付属しているため、追加のレイヤ、レシピ、`local.conf` エントリは必要ありません。

すぐに使用可能なアルゴリズム:

カテゴリ	アルゴリズム
KEM	ML-KEM-512, ML-KEM-768, ML-KEM-1024
ハイブリッド KEM (TLS)	X25519MLKEM768, X448MLKEM1024, SecP256r1MLKEM768, SecP384r1MLKEM1024

カテゴリ	アルゴリズム
シグネチャ	ML-DSA-44、ML-DSA-65、ML-DSA-87
ハッシュベースのシグネチャ	SLH-DSA (12 のバリエーション — SHA-2 と SHAKE、小規模 / 高速のトレードオフ)

TLS コマンド (`openssl s_server`、`openssl s_client`)、鍵生成、署名、証明書作成、TLS ハンドシェイクの詳細なチュートリアルについては、次を参照してください。

- [3.4.6.ポスト量子暗号 — AM62Lx SDK ドキュメント](#)
- [openssl-s_server\(1\)](#)、[openssl-s_client\(1\)](#)、[openssl-genpkey\(1\)](#)

注

OpenSSL ネイティブでカバーされていないもの:

ベース SDK に組み込まれていないものには、`oqs-provider (x25519_mlkem512, p384_mlkem768)` のハイブリッド KEM、非 NIST アルゴリズム (FrodoKEM、BIKE)、ステートフル ハッシュベース シグネチャ (LMS、XMSS)、wolfSSL NEON アクセラレーション ネイティブ API があります。ステップ Y1 から Y2 と以下に示すオプション A/B は、これらの使用事例に合わせて SDK を拡張します。

2.3 liboqs/oqs-provider 統合

このパスは、`liboqs` のハイブリッド KEM、FrodoKEM、および `oqs-provider` を介した BIKE を使用して OpenSSL を拡張します。2 つのステップ:

ステップ Y1 —メタのクローン作成

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

ステップ Y2 — local.conf での OQS の有効化

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

`DISTRO_FEATURES:append = " oqs"` 必須。`meta-pqc` の OQS OpenSSL プロバイダの構成インストール、`oqs-provider RDEPEND`、および PQC 対応 `openssl/ca-certificates bbbadds` をゲートします。

警告

`liboqs` は `clang-native/libcxx-native` でビルドします

`meta-oqs` の `liboqs_0.15.0.bb` (および `liboqs-rust_0.10.1.bb`) は `TOOLCHAIN = "clang"` を設定します。これは通常、レシピごとに `clang` ツールチェーンを要求するストック TI SDK ビルドの最初のもので、これにより `oe-core` の `clang-native.bbclass` がプルされます。これは多くの場合、ビルド全体の初回で、**clang-native**、**compiler-rt-native**、**libcxx-native** (`libc++/libc++abi` の LLVM のブートストラップ) に依存します。

`libcxx-native` のネイティブ コンパイラ ABI のチェック中にビルドのサンドボックス化された `hosttools` 環境で `-lstdc++` を解決できない場合、`libcxx-native` の `do_configure` は「cannot find-lstdc++」により失敗します。そのようなファイルやディレクトリは存在せず、**liboqs** はビルドされません。**liboqs** 自体には **clang** 固有の要件はなく、SIMD チューニング オプション (`OQS_USE_AVX2_INSTRUCTIONS`、`OQS_USE_ARM_NEON_INSTRUCTIONS` など) は、GCC が同じように処理するプレーンな `-m <isa>` スタイルのフラグです。この修正は、コントロール レイヤで `bappend` を使用して、デフォルトのディストリビューション ツールチェーンに強制的に戻すというものです (`meta-tisdk` など)。

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

2.4 ビルド エラーからの回復

ビルド中にレシピで障害が発生した場合は、強制的にレシピをクリーンに再ビルドします。

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

これで問題が解決しない場合は、依存するレシピ (例えば、リンクまたはビルド対象のライブラリやカーネル) も `cleansstate` します。

例:現在の **OpenSSL** に対する **libwebsockets** 障害:

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

このようなビルドの障害のほとんどは、ソースのバグではなく、ビルド状態が古い、または整合性がないこと、つまり、(例えば、バージョンのバンプ後、設定変更後、ビルドの中断後に) レシピの永続的な `work/` ディレクトリまたは `sstate-cache` エントリと依存する項目との同期が失われたことが原因です。`cleansstate` と再ビルドは、すべての場合において信頼できる修正法です。

2.5 wolfSSL の統合

このパスは PQC で wolfSSL を拡張し、OpenSSL にフックします。2 つの統合オプションがサポートされており、どちらも同じ PQC アルゴリズムを有効にします。

2.5.1 パスの選択

2 つのパスがサポートされています。どちらのパスも同じ PQC アルゴリズムを有効にします。

要件	オプション A:meta-wolfssl	オプション B:meta-oe wolfssl
wolfSSL ライブラリ (libwolfssl.so)	あり	あり
PQC アルゴリズム (ML-KEM、ML-DSA、LMS、XMSS)	あり	あり
wolfcryptbenchmark バイナリがターゲットにあること	あり	なし
wolfcrypttest バイナリがターゲットにあること	あり	なし
wolfprovider (OpenSSL プロバイダ)	あり	なし
追加のレイヤ クローンが必要	あり (meta-wolfssl)	なし
BBMASK エントリが必要	あり (5 エントリ)	なし

オプション A は、ボード上でベンチマークまたはテスト バイナリが必要な場合に使用します。オプション B は、アプリケーションでライブラリだけがが必要な場合に使用します。

2.5.2 ステップ A1 — オプション A:meta-wolfssl

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

`bblayers.conf` に以下を追加します。

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

`local.conf` に以下を追加します。

```
# 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"
```

注

セクション 2.7 では **wolfprovider** が必要です

セクション 2.7 の統合型 3 プロバイダ OpenSSL 構成を有効にするには、**wolfprovider** が **IMAGE_INSTALL** になければなりません。これは **meta-wolfssl** により提供されており、**libwolfprov.so** が含まれています。これがなければ、**providers.conf** の **[wolfprov_sect]** ブロックは正しくロードされません。

wolfSSL のカスタマイズはすべて **meta-wolfssl/recipes-wolfssl/wolfssl_%.bbappend** で行います (バージョン固定の **wolfssl_5.9.1.bbappend** は同じように動作し、bitbake はレシピにマッチします)。

```
# 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 は、Cortex-A53 でハードウェア AES、SHA-256、SHA-512、および **NEON Keccak (SHAKE)** に対して wolfSSL の ARM アセンブリ パスを有効にします。詳細なインパクト分析については、[セクション 4](#) を参照してください。

警告

armasm + wolfprovider ヘッダーの競合 (aarch64)

--enable-armasm と **--enable-opensslcoexist** がどちらも有効である場合 (wolfprovider には **--enable-opensslcoexist** が必要)、インストールされた wolfSSL の **wolfssl/wolfcrypt/aes.h** は、シグネチャ **void(const unsigned char*, word32, unsigned char*)** を使用して **#if defined (WOLFSSL_ARMASM)** で内部シンボル **AES_set_encrypt_key** を宣言します。**options.h** に **OPENSSL_COEXIST** が設定されている場合、wolfSSL は意図的に **AES_BLOCK_SIZE** を省略するので、wolfprovider の **internal.h** は **AES_set_encrypt_key** を **int(const unsigned char*, const int, AES_KEY*)** として宣言する OpenSSL の **aes.h** を取り込みます。両方の宣言が同じ変換単位に入り、タイプ競合のコンパイラ エラーが発生します。

修正は **meta-wolfssl/recipes-wolfssl/wolfssl_%.bbappend** にあり、**do_install: append: aarch64** フックは wolfSSL 内部宣言の名前をインストールされたヘッダ内で **wolfssl_arm_AES_set_encrypt_key** に変更するために **sed** を使用します。**wolfSSL.so** は既に **WOLFSSL_LOCAL (GCC 非公開)** とマークされたシンボルでコンパイルされており、wolfprovider は **wc_AesSetKey** を通して wolfSSL AES を呼び出し、決してこの内部 **armasm** ヘルパーを直接呼び出すことはないため、これは安全な方法です。

```
# 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}
```

これにより **do_install** 出力が変更されるため、**wolfssl** はインストール ステップを再ビルドする必要があり、**wolfprovider** は名前を変更したヘッダーに対して再ビルドする必要があります。

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

注

ベンチマークのブロック サイズ (-DBENCH_EMBEDDED)

Yocto wolfcryptbenchmark バイナリは、**-DBENCH_EMBEDDED** (1 KB ブロック サイズ) と **-DUSE_CERT_BUFFERS_2048** (RAM 内証明書) でビルドされます。これは、以下のことを意味しています。

- 対称暗号スループット (AES-CBC、AES-GCM、ChaCha20) は 1KB ブロックで測定され、L1 キャッシュ レジデンシーにより 1MB ブロック実行よりも数値が高く、組込み TLS の使用事例 (TLS レコード ≤ 16KB、ネットワーク パケット約 1.5KB) に適しています。**Cortex-A53** の L1 キャッシュは **32KB** です。1MB ベンチマーク バッファによりキャッシュ スラッシュが発生し、組込みワークロードでは現実的ではないスループット値が生成されます。
- **PQC ops/sec** (ML-KEM、ML-DSA、LMS、XMSS) は影響を受けません。ブロック サイズは固定サイズの keygen/encaps/decaps/sign/verify 操作とは無関係です。

ベンチマーク ヘッダーが表示されます (ブロック バイト **1024**、最小 **1.0** 秒)。以下が想定されます。

既知の flaky メカニズム: **-DBENCH_EMBEDDED/-DUSE_CERT_BUFFERS_2048** の定義 (および類似の **wolfcrypttest** の定義: **-DNO_WRITE_TEMP_FILES -DUSE_CERT_BUFFERS_2048 -DUSE_CERT_BUFFERS_256 -DWOLFSSL_RSA_KEY_CHECK**) は、**meta-wolfssl** の条件付きで必要な **.inc** ファイルを介して **TARGET_CFLAGS** に追加されています。このメカニズムは、一部のビルドではコンパイラに到達せず、他のビルドではレシピの変更なしで正常に動作します。**wolfcrypttest** の **do_package_qa** が **buildpaths** エラーで失敗した場合 (インストールされたバイナリに組み込まれた絶対ビルド ホストパスであり、同じ欠落した定義は、バイナリがターゲット上のいくつかのサブテストで存在しないビルドホストパスを開こうとすることを意味するため、単なる表面的なものではない)、**bitbake-c cleansstate wolfssl** を再実行し、**grep USE_CERT_BUFFERS_2048 <wolfssl work dir>/temp/log.do_compile** をチェックします。それが存在しない場合、定義はその時間を伝播せず、再試行すること (またはこのネストされた **.inc** メカニズムに依存するのではなく、自身のコントロールレイヤの **bbappend** を介して同じ定義を強制適用すること) が信頼できる修正です。

2.5.3 ステップ A2 — オプション B :meta-oe wolfssl (ライブラリのみ)

meta-openembedded/meta-networking meta-openembedded/meta-networking が TI SDK **bblayers.conf** にすでに記述されています。追加のクローンは必要ありません。**BBMASK** エントリは必要ありません。

制限: **libwolfssl.so** のみがインストールされています。**wolfcryptbenchmark** および **wolfcrypttest** バイナリはありません。**meta-oe** レシピには **wolfssl 5.9.0** (**meta-wolfssl** のように **5.9.1** ではない) が付属しています。セクション 1.1 の「プラットフォーム」表には、オプション A の構成が反映されています。

local.conf に以下を追加します。

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

meta-oe で直接 PQC **bbappend** を作成します (これにより、PQC サポートを追加するために **meta-oe wolfssl** レシピが変更されます)。

```
# 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

meta-oe を直接変更。このアプローチでは、**meta-oe** レイヤを直接編集します。プライベート SDK では機能しますが、次の点に注意してください。

- **meta-oe** が更新またはリフレッシュされると、変更内容が失われる可能性があります
- これは、本番環境のワークフローで推奨される Yocto の運用ではありません
- 共有または永続的な作業の場合は、代わりにオプション A (**meta-wolfssl**) を使用するか、専用の **meta-pqc** レイヤを作成することを検討してください

meta-oe を直接変更したくない場合は、上記のオプション A を参照するか、**bblayers.conf** の **meta-oe** の後にある独自の制御レイヤ (**meta-tisdk**、**meta-ti**、**meta-custom** など) に **bbappend** を作成してください。

2.6 ステップ B — ビルド

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

注

既知の問題: ビルドが中断された状態では、レシピ独自のソース チェックアウトに不具合が残る可能性があります

bitbake の実行がタスクの途中で中断された場合 (Ctrl-C)、このアプリケーション ノートに記載されているものとは別に、一部のレシピにおいて、次の実行で構成途中のソース チェックアウトが残る可能性があります。

kernel-selftest で発生する症状: ソースツリーがクリーンでない状態で **do_configure** が失敗します。

kernel-selftest 自身の解凍されたソース ディレクトリで '**make ARCH=arm64 mrproper**' を実行してください (共有カーネル ビルド ツリーではありません。修正のために、カーネルを再ビルドする必要はありません)。修正は影響を受けるレシピのみに適用されます。

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

2.7 ステップ C — OpenSSL プロバイダの構成

3 つのプロバイダは **/etc/ssl/openssl.cnf.d/providers.conf** で構成されます。デフォルト イメージには、**openssl-oqs.cnf** と競合する **wolfprovider.conf** および **wolfprovider-fips.conf** が付属しています ([**openssl_init**] および [**provider_sect**] セクション名が重複すると、再帰的なセクション参照エラーが発生します)。これらを統合構成に置き換えます。

```
# 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
```

注

競合の根本原因:

/etc/ssl/openssl.cnf が **.include** ディレクティブを使用している場合、OpenSSL の構成パーサーはインクルードされたファイルの処理後にセクション コンテキストを復元しません。**wolfprovider-fips.conf** が **[algorithm_sect]** で終わる場合、次の **.include** が処理されるときにパーサーはそのセクションに残ります。これにより、次のファイルの **openssl_conf=openssl_init** が **[algorithm_sect]** 内部のキーとして解析され、**alg_module_init: unknown option** をトリガします。修正するには、単一の統合ファイル内でセクション名を一意にします。

3 つのプロバイダすべてがロードされることを確認:

```
openssl list -providers
```

想定出力:

```
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 性能ベンチマーク

注

ベンチマークを再現するための前提条件パッケージ:

このセクションのコマンドには、**IMAGE_INSTALL** に次のパッケージが必要です (セクション 2.3.2 および 2.5.2 を参照)。

- **liboqs-tests** liboqs-tests → **oqs_speed_kem**, **oqs_speed_sig** を /usr/bin/ にインストールします /usr/bin/
- **wolfcryptbenchmark** wolfcryptbenchmark → /usr/bin/wolfcryptbenchmark をインストールします (meta-wolfssl bbappend を介して。wolfssl ビルドで **--enable-crypttests** をトリガーします)

これらのパッケージがない場合、バイナリは **rootfs** 上に存在しません。速度ベンチマークを実行するには、**liboqs** ライブラリと **oqsprovider.so** をインストールするだけでは不十分です。

すべてのベンチマークは、AM62L3 EVM、Cortex-A53 上で、約 1.4GHz、Arago 2025.01 で動作します。wolfSSL 番号は **--enable-armasm** (NEON Keccak + ハードウェア AES/SHA-2 アセンブリがアクティブ) を使用します。

すべてのベンチマークは、AM62L3 EVM、Cortex-A53 上で、約 1.4GHz、Arago 2025.01 で動作します。wolfSSL 番号は **--enable-armasm** (NEON Keccak + ハードウェア AES/SHA-2 アセンブリがアクティブ) を使用します。

3.1 ML-KEM — すべての実装

```
# 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
```

アルゴリズム	OpenSSL ネイティブ keygen/s	liboqs (NEON) keygen/s	wolfSSL keygen/s (+armasm)	OpenSSL encaps/s	liboqs encaps/s	wolfSSL encaps/s (+armasm)	OpenSSL decaps/s	liboqs decaps/s	wolfSSL decaps/s (+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

注

wolfSSL(+armasm) が ML-KEM に基づく liboqs NEON より優位に立つ理由:

--enable-armasm を使用すると、wolfSSL により、SHAKE/Keccak (**sha3_asm.S**) を含むすべての暗号プリミティブに対して aarch64 アセンブリが有効化されます。ML-KEM の性能は **SHAKE-128/256 (Keccak)** によって決まるため、NEON Keccak アセンブリにより、wolfSSL が純粋な C のベースラインより **60~100% 向上**し、ML-KEM-512 に対して liboqs を **keygen** で **47%**、**encaps** で **67%** 上回る結果が得られます。

liboqs の NEON NTT パスにより多項式算術が最適化されますが、wolfSSL の NEON keccak の速度はそれよりも優れています。**--enable-armasm** を使用しない場合、wolfSSL ML-KEM では純粋な C Keccak が実行され、liboqs の方が優位に立ちます。前後の比較については、[セクション 3.5](#) を参照してください。

3.2 ML-DSA — すべての実装

```
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
```

アルゴリズム	OpenSSL keygen/s	liboqs keygen/s	wolfSSL keygen/s (+armasm)	OpenSSL sign/s	liboqs sign/s	wolfSSL sign/s (+armasm)	OpenSSL verify/s	liboqs verify/s	wolfSSL verify/s (+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

注

wolfSSL が ML-DSA に基づく liboqs NEON より優位に立つ理由:

liboqs 0.15.0 には **ML-DSA 専用の aarch64 NTT** はなく、x86 用の AVX2 パスのみ存在します (OQS_ENABLE_SIG_ml_dsa_65_avx2 は無効で、_aarch64 も _neon バリエーションも存在しません)。liboqs ML-DSA における NEON の利点は、NTT 多項式算術ではなく、NEON で促進される SHAKE (OQS_USE_ARM_SHA3_INSTRUCTIONS) のみに由来します。wolfSSL の C NTT for Diliithium/ML-DSA はより最適化されていて、署名時に liboqs を約 3 倍、keygen/verify 時に約 2 倍上回ります。

3.3 FrodoKEM (liboqs のみ)

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

アルゴリズム	keygen/s	encaps/s	decaps/s	備考
FrodoKEM-640-AES	310	270	278	ハードウェア AES がアクティブ
FrodoKEM-976-AES	145	120	126	ハードウェア AES がアクティブ
FrodoKEM-1344-AES	81	67	67	ハードウェア AES がアクティブ

FrodoKEM は、行列演算が大規模なため、時間が ML-KEM の約 25 ~ 40 倍かかりますが、ハードウェア AES (OQS_USE_ARM_AES_INSTRUCTIONS=ON) の恩恵を直接受けます。

3.4 ハイブリッド KEM — TLS オーバーヘッド

```
# 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

wolfprovider が共ロードされている場合の **p384_mlkem768** のセグメンテーション フォルト

統合型 **providers.conf** (3 つのプロバイダすべてがアクティブ) を使用して **openssl speed ... p384_mlkem768** を実行すると、**oqs-provider 0.11.0** でセグメンテーション フォルトが発生します。根本原因: **wolfprovider** と **oqsprovider** の **P-384** 鍵交換パス間の相互作用。回避方法は、上記のように **OPENSSL_CONF=/dev/null** で **providers.conf** をバイパスし、**default** と **oqsprovider** のみを明示的にロードすることです。**x25519_mlkem512** は影響を受けず、3 つのプロバイダすべてがロードされた状態で正しく実行されます。

アルゴリズム	プロバイダ	keygen/s	encaps/s	decaps/s	ボトルネック
ML-KEM-768	デフォルト (OpenSSL)	3,130	3,963	2,668	純粋な PQC
X25519MLKEM768	デフォルト (OpenSSL)	1,653	1,036	1,211	X25519 ECDH
x25519_mlkem512	oqsprovider	約 2,600	約 1,185	約 1,200	X25519 ECDH
p384_mlkem768	oqsprovider	約 21	約 63	約 108	P-384 スカラー倍算 (鍵生成 47ms)

ハイブリッド バリエーションでは PQC 成分は無視できるものであり、古典的な ECDH が完全に優勢となります。

3.5 wolfSSL の性能に対する armasm の影響

--enable-armasm aarch64 では、ハードウェア AES、ハードウェア SHA-256/SHA-512、および **NEON Keccak/SHAKE** (sha3_asm.s) の wolfSSL アセンブリを有効にします。次の表に、PQC アルゴリズムの性能の差を示します。

アルゴリズム	動作	armasm なし (ops/s)	armasm あり (ops/s)	改善
ML-KEM-512	keygen	11,896	18,897	+59%
ML-KEM-512	encaps	10,316	18,355	+78%
ML-KEM-512	decaps	7,258	12,876	+77%
ML-KEM-768	keygen	6,145	11,881	+93%
ML-KEM-768	encaps	6,135	11,727	+91%
ML-KEM-768	decaps	4,580	8,521	+86%
ML-KEM-1024	keygen	3,823	7,820	+105%
ML-KEM-1024	encaps	3,884	6,423	+65%
ML-KEM-1024	decaps	3,061	5,572	+82%
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 署名は除去サンプリングを使用します。反復カウントは確率的であり、 $\pm 10\%$ ~ $\pm 15\%$ のベンチマークの変動が発生します。ML-DSA-44/65 で観測される署名回帰はノイズ内にあります。

注

ML-KEM が armasm から最大のメリットを享受する理由:

ML-KEM は SHAKE-128/256 (Keccak) を主要なプリミティブとして使用しており、鍵生成は約 60% の Keccak、カプセル化は約 70% の Keccak です。wolfSSL の *sha3_asm.s* は NEON で最適化された Keccak-f [1600] 置換により、すべての ML-KEM の動作を 60% ~ 100% 高速化します。ML-DSA では SHAKE の使用が控えめであるため (支配的な NTT 多項式算術ではなく、ハッシュ用)、SHAKE の高速化による全体的な改善は小規模なものとなります。

4 NEON および HW アクセラレーション分析

4.1 実装ごとのアクセラレーション ステータス

実装	ML-KEM NTT	ML-DSA NTT	SHAKE/Keccak	AES	SHA-2
OpenSSL 3.5 のデフォルト	純粋な C	純粋な C	NEON keccak1600 asm	HW	HW
liboqs 0.15.0 (Yocto)	aarch64 NEON NTT✓	純粋な C (aarch64 パスなし)	NEON Keccak (XKCP)✓	HW✓	HW✓
wolfSSL 5.9.1 (armasm なし)	最適化 C	最適化 C	純粋な C	純粋な C	純粋な C
wolfSSL 5.9.1 (+armasm)	最適化 C	最適化 C	NEON Keccak asm ✓	HW asm ✓	HW asm ✓

主な所見:

- wolfSSL --enable-armasm は、ハードウェア AES/SHA-2 に加えて **NEON Keccak (sha3_asm.S)** を有効にします。これは、ML-KEM を 60% ~ 100% 高速化するプライマリドライバです
- liboqs には **ML-KEM 専用の aarch64 NEON NTT** がありますが、SHAKE に対する wolfSSL の NEON Keccak の優位性はそれを上回り、wolfSSL+armasm は全体的に ML-KEM より優位に立っています
- liboqs には **ML-DSA 用の aarch64 NTT** がありません (AVX2 は liboqs 0.15.0 でのみ)。— wolfSSL の最適化 C は、armasm とは関係なく、ML-DSA より優位に立っています
- liboqs も wolfSSL も SHA-3 ハードウェアを使用することはできません (Cortex-A53 には対応していません)。SHAKE には NEON Keccak が最適です

4.2 アルゴリズムによる推奨

使用事例	推奨パス	理由
ML-KEM 鍵交換 (TLS)	wolfSSL ネイティブ API (+armasm)	最高速: NEON Keccak asm — liboqs NEON NTT よりも 47% ~ 67% 高速
ML-DSA シグネチャ	wolfSSL ネイティブ API	最高速: liboqs より 3 倍高速な署名
TLS 用ハイブリッド KEM (X25519+ML-KEM)	OpenSSL ネイティブ X25519MLKEM768	最小限のオーバーヘッドで、追加のプロバイダは不要
FrodoKEM	liboqs (oqs-provider 経由)	利用可能な実装のみ
LMS/XMSS ステートフル シグネチャ	wolfSSL/liboqs ネイティブ API	完全な実装 (keygen+sign+verify)
スタンドアロン PQC アプリ (OpenSSL なし)	wolfSSL ネイティブ C API (+armasm)	全体的な PQC 性能が最高で、EVP オーバーヘッドはなし

5 アルゴリズム可用性の概要

アルゴリズム	OpenSSL 3.5 ネイティブ	oqs-provider (liboqs)	liboqs ネイティブ	wolfprovider	wolfSSL ネイティブ
ML-KEM-512/768/1024	✓	ハイブリッド バリエーションのみ†	✓	— ‡	✓
ML-DSA-44/65/87	✓	OID 競合 — ブロック†	✓	— ‡	✓
SLH-DSA (すべて 12)	✓	—	✓	—	—
X25519MLKEM768、X448MLKEM1024	✓ (TLS ハイブリッド)	—	—	—	—
x25519_mlkem512、p384_mlkem768	—	✓	—	—	—
FrodoKEM-640/976/1344	—	✓	✓	—	—
BIKE-L1/L3/L5	—	✓	✓	—	—
LMS/HSS	検証のみ	—	✓	—	✓
XMSS/XMSS^MT	—	—	✓	—	✓

† OpenSSL 3.5 上の oqs-provider v0.11.0 は、OID とネイティブ実装との競合を回避するために、スタンドアロン ML-KEM および ML-DSA を自動抑制します。

‡ wolfProvider 独自の README によれば、PQC のサポートには、**OpenSSL 3.6+** および wolfSSL master/v5.9.2-stable+ と明示的な `--enable-pqc` ビルド フラグが必要です。

この SDK では **OpenSSL 3.5.5** に固定されるため、wolfSSL 独自の PQC サポートに関係なく、wolfprovider の PQC パスは利用できません。

wolfSSL の ML-KEM/ML-DSA は実際に機能しますが、ネイティブ **wolfCrypt C API** (`wolfcryptbenchmark`/`wolfcrypttest`) からのみアクセスでき、wolfprovider や openssl CLI からはアクセスできません。

6 参考資料

1. テキサス インストルメンツ、[AM62Lx SDK — ポスト量子暗号](#)
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 の Limitations の規定](#)
9. 米国国立標準技術研究所、[NIST FIPS 203 — ML-KEM](#)
10. 米国国立標準技術研究所、[NIST FIPS 204 — ML-DSA](#)
11. 米国国立標準技術研究所、[NIST FIPS 205 — SLH-DSA](#)
12. 米国国立標準技術研究所、[NIST SP 800-208 — LMS/XMSS](#)
13. Yocto Project、[Yocto 6.0 wrynose 移行ガイド](#)
14. テキサス インストルメンツ、[AM62LX SDK — Building the SDK with Yocto](#)
15. wolfSSL、[wolfProvider README — Post-Quantum \(NIST FIPS 203/204\)](#)

重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含みいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとし、TI は一切の責任を拒否します。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、[TI の販売条件](#)、[TI の総合的な品質ガイドライン](#)、[ti.com](#) または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

Copyright © 2026, Texas Instruments Incorporated

最終更新日：2025 年 10 月