



wolfCOSE Manual

2026-10-16

Contents

1	wolfCOSE	8
1.1	Standards	8
1.2	Key Features	8
1.3	Documentation	9
1.4	Supported Message Types	9
1.5	Quick Links	9
1.6	License	9
2	Getting Started	10
2.1	Prerequisites	10
2.1.1	wolfSSL Installation	10
2.1.2	Minimal Builds	10
2.1.3	Feature to wolfSSL Flag Mapping	10
2.2	Building wolfCOSE	11
2.2.1	wolfSSL Discovery	11
2.2.2	Build Targets	11
2.3	Quick Start: Sign and Verify	12
2.4	Quick Start: Encrypt and Decrypt	13
2.5	Experimental COSE-HPKE P0	14
2.6	Quick Start: Post-Quantum Signing (ML-DSA)	15
2.7	CLI Tool	16
2.8	Examples Directory	17
2.8.1	Comprehensive Tests (examples/comprehensive/)	17
2.8.2	Real-World Scenarios (examples/scenarios/)	17
2.9	Strict Decoding (RFC 8949 Preferred Serialization)	17
2.10	Cross-Compilation	18
2.11	Stack Budget	19
2.12	Next Steps	19
3	COSE Message Types	20
3.1	COSE_Sign1 — single-signer signature (RFC 9052 Sec. 4.2)	20
3.2	COSE_Sign — multi-signer signature (RFC 9052 Sec. 4.1)	20
3.3	RFC 9338 countersignatures	21
3.3.1	Relationship to SCITT	22
3.4	COSE_Encrypt0 — single-recipient AEAD (RFC 9052 Sec. 5.2)	22
3.5	COSE_Encrypt — multi-recipient AEAD (RFC 9052 Sec. 5.1)	22
3.6	COSE_Mac0 — single-recipient MAC (RFC 9052 Sec. 6.2)	23
3.7	COSE_Mac — multi-recipient MAC (RFC 9052 Sec. 6.1)	23
3.8	Detached payloads	23
3.9	Compile-time stripping	23
3.10	See also	24
4	Supported Algorithms	25
4.1	COSE_Sign1 (Digital Signatures)	25
4.1.1	Migrating from RFC 9053 Signature IDs	26
4.1.2	ML-DSA Signature Sizes	26
4.2	COSE_Encrypt0 (Authenticated Encryption)	27
4.2.1	AES-CCM Naming Convention	27
4.3	COSE_Mac0 (Message Authentication)	27
4.4	Key Distribution (Multi-Recipient)	28
4.5	Key Types	28
4.6	Curves	29

4.7	COSE Tags	29
4.8	Algorithm Constants in Code	29
4.9	Roadmap	30
4.10	See Also	30
5	API Reference	31
5.1	Table of Contents	31
5.2	Data Structures	31
5.2.1	WOLFCOSE_KEY	31
5.2.2	WOLFCOSE_HDR	31
5.2.3	WOLFCOSE_SIGNATURE	32
5.2.4	WOLFCOSE_COUNTERSIGNATURE	32
5.2.5	WOLFCOSE_RECIPIENT	32
5.2.6	WOLFCOSE_CBOR_CTX	33
5.3	COSE_Key API	33
5.3.1	wc_CoseKey_Init	33
5.3.2	wc_CoseKey_Free	33
5.3.3	wc_CoseKey_SetEcc	33
5.3.4	wc_CoseKey_SetEd25519	34
5.3.5	wc_CoseKey_SetEd448	34
5.3.6	wc_CoseKey_SetMIDsa	34
5.3.7	wc_CoseKey_SetMIDsa_ex	34
5.3.8	wc_CoseKey_SetRsa	35
5.3.9	wc_CoseKey_SetLms	35
5.3.10	wc_CoseKey_SetSymmetric	35
5.3.11	wc_CoseKey_Encode	36
5.3.12	wc_CoseKey_Encode_ex	36
5.3.13	wc_CoseKey_EncodeEccRaw	36
5.3.14	wc_CoseKey_EncodeSize / wc_CoseKey_EncodeSize_ex	37
5.3.15	wc_CoseKey_Decode	37
5.3.16	wc_CoseKey_PeekInfo	38
5.4	COSE_Sign1 API	39
5.4.1	wc_CoseSign1_Sign	39
5.4.2	wc_CoseSign1_Sign_ex	39
5.4.3	wc_CoseSign1_SignSize_ex	40
5.4.4	wc_CoseSign1_Verify	40
5.5	COSE Countersignature API	41
5.5.1	wc_Cose_AddCounterSignature	41
5.5.2	wc_Cose_AddCounterSignature0	41
5.5.3	wc_Cose_VerifyCounterSignature	42
5.5.4	wc_Cose_VerifyCounterSignature0	42
5.6	COSE_Encrypt0 API	42
5.6.1	wc_CoseEncrypt0_Encrypt	42
5.6.2	wc_CoseEncrypt0_Decrypt	43
5.7	COSE_Mac0 API	43
5.7.1	wc_CoseMac0_Create	43
5.7.2	wc_CoseMac0_Verify	44
5.8	COSE_Sign API (Multi-Signer)	44
5.8.1	wc_CoseSign_Sign	44
5.8.2	wc_CoseSign_Verify	44
5.9	COSE_Encrypt API (Multi-Recipient)	45
5.9.1	wc_CoseEncrypt_Encrypt	45
5.9.2	wc_CoseEncrypt_Decrypt	45
5.10	COSE_Mac API (Multi-Recipient)	46

5.10.1 wc_CoseMac_Create	46
5.10.2 wc_CoseMac_Verify	46
5.11 PSA/EAT API	47
5.11.1 wc_CoseEatPsaToken_Verify	47
5.11.2 Other PSA/EAT entry points	47
5.12 CBOR API	48
5.12.1 Context Setup	48
5.12.2 Encoding Functions	48
5.12.3 Decoding Functions	49
5.12.4 wc_CBOR_SkipItem	49
5.12.5 wc_CBOR_DecodeLabel	50
5.13 Error Codes	51
5.14 See Also	51
6 Configuration Macros	52
6.1 Experimental Features	52
6.2 Lean Configuration Layer	53
6.3 PSA/EAT Attestation Gates	53
6.4 Algorithm Gates	54
6.4.1 Deprecated Algorithm IDs (RFC 9864)	55
6.4.2 ECDSA Nonce Policy	56
6.5 Message Type Gates	56
6.5.1 COSE Countersignatures	56
6.5.2 COSE_Sign1 (Single Signer)	57
6.5.3 COSE_Encrypt0 (Symmetric Encryption)	57
6.5.4 COSE_Mac0 (Symmetric MAC)	57
6.5.5 COSE_Sign (Multi-Signer)	58
6.5.6 COSE_Encrypt (Multi-Recipient)	58
6.5.7 COSE_Mac (Multi-Recipient)	58
6.6 Key Distribution Gates	58
6.7 COSE-HPKE (experimental)	59
6.8 Delegated Signing	60
6.9 CBOR Layer Gates	62
6.10 COSE_Key Gates	62
6.11 Size Configuration	62
6.11.1 WOLFCOSE_MIN_BUFFERS	62
6.12 Tuning for Size	63
6.13 Tuning for Speed	63
6.14 Build Profiles	64
6.14.1 Footprint	64
6.14.2 WOLFCOSE_LEAN_VERIFY — minimal verify-only	64
6.14.3 WOLFCOSE_LEAN_MLDSA — lean post-quantum sign + verify	64
6.14.4 WOLFCOSE_LEAN_VERIFY_MLDSA — minimal post-quantum verify-only	65
6.14.5 WOLFCOSE_LEAN_LMS — lean stateful hash-based sign + verify	65
6.14.6 WOLFCOSE_LEAN_VERIFY_LMS — minimal hash-based verify-only	65
6.15 Example Build Configurations	65
6.15.1 Sign-Only Build (Minimal)	65
6.15.2 Verify-Only Build	65
6.15.3 Sign1-Only Build (Smallest)	65
6.15.4 No Multi-Recipient Support	66
6.16 wolfSSL Dependencies	66
6.17 Test and Example Gates	66
6.17.1 Comprehensive Test Gates	66
6.17.2 Scenario Example Gates	67

6.18 See Also	67
7 Project Structure	68
7.1 Directory Layout	68
7.2 Core Library Files	69
7.2.1 include/wolfcose/wolfcose.h	69
7.2.2 include/wolfcose/visibility.h	69
7.2.3 COSE message modules (src/wolfcose_*.c)	69
7.2.4 src/wolfcose_cbor.c	70
7.2.5 src/wolfcose_internal.h	70
7.3 Build Outputs	70
7.4 Test Files	70
7.4.1 tests/test_cbor.c	70
7.4.2 tests/test_cose.c	70
7.4.3 tests/test_interop.c	70
7.4.4 tests/test_main.c	70
7.4.5 tests/force_failure.c / force_failure.h	71
7.5 CLI Tool	71
7.5.1 tools/wolfcose_tool.c	71
7.6 Examples	71
7.6.1 Basic Demos	71
7.6.2 Comprehensive Tests (examples/comprehensive/)	71
7.6.3 Scenario Examples (examples/scenarios/)	71
7.7 Integration Notes	72
7.7.1 Minimal Integration	72
7.7.2 CBOR-Only Integration	72
7.7.3 Production Builds	72
7.8 See Also	72
8 PSA Attestation and EAT Tokens	73
8.1 Select only what is needed	73
8.2 API	75
8.2.1 PSA client handoff	75
8.3 Profile behavior	77
8.4 COSE and trust boundary	77
8.5 Validation and interoperability	78
9 MISRA C Compliance	79
9.1 Coverage	79
9.1.1 MISRA C:2012	79
9.1.2 MISRA C:2023	79
9.2 Coverage Summary	80
9.3 MISRA C:2012 Rule Enforcement	80
9.3.1 Approved deviations	80
9.4 MISRA C:2023 Deviations (clang-tidy)	81
9.4.1 bugprone-branch-clone: Identical Consecutive Switch Branches	81
9.4.2 bugprone-easily-swappable-parameters: Adjacent Parameters of Similar Types	81
9.5 Examples and Test Code	82
9.5.1 Rule 21.6: Standard I/O (and 17.7 on its return value)	82
9.5.2 Rule 15.5: Multiple return / single point of exit	82
9.5.3 Rule 2.5: Unused macro definitions	82
9.5.4 Rules 8.6 / 5.9 / 8.9: External/internal identifier definitions	82
9.6 Fully Compliant Rules (Notable)	82
10 Footprint and Performance	84

10.1 Library size (glue)	84
10.2 Total footprint with wolfCrypt	84
10.3 Build-profile footprint	84
10.3.1 Post-quantum at the same cost (ML-DSA-44, FIPS 204)	85
10.4 Speed (Intel i9-11950H, x86_64)	85
10.5 On a real MCU: STM32H563 (Cortex-M33 @ 250 MHz)	85
10.6 Method and versions	85
10.7 See Also	85
11 Testing	86
11.1 Running Tests	86
11.1.1 Release Qualification	86
11.1.2 Basic Unit Tests	86
11.1.3 PSA/EAT Attestation Tests	86
11.1.4 Deprecated Algorithm ID Tests	87
11.1.5 CLI Tool Tests	87
11.1.6 Experimental COSE-HPKE Tests	87
11.1.7 Comprehensive Algorithm Tests	88
11.1.8 Scenario Examples	88
11.1.9 Interoperability (t_cose)	88
11.1.10 Complete upstream t_cose suite	89
11.1.11 Interoperability (go-cose)	89
11.1.12 Interoperability (python-cwt)	89
11.1.13 Interoperability (Rust coset)	89
11.1.14 COSE WG Examples vectors	90
11.1.15 PSA attestation-token acceptance	90
11.2 Code Coverage	90
11.2.1 Running Coverage	90
11.2.2 Coverage Targets	90
11.2.3 Coverage with Failure Injection	90
11.3 Force Failure Testing	90
11.3.1 Production Builds	91
11.4 CI Pipeline	91
11.4.1 Build and Test Matrix	91
11.4.2 Test Stages	91
11.4.3 Memory and Stack Bounds	91
11.4.4 Lean and Post-Quantum Builds	91
11.4.5 Static Analysis	92
11.4.6 Coverity Scan	92
11.5 Test File Structure	92
11.5.1 Test Categories in test_cose.c	92
11.6 Test Vectors	93
11.7 See Also	93
12 STM32Cube	94
12.1 Prerequisites	94
12.2 Install the packs	94
12.3 Add wolfCOSE to a project	94
12.4 Configure algorithms	94
12.5 Run on a device	94
12.6 Notes	95
13 Experimental Features	96
13.1 Customer Summary	96
13.2 COSE-HPKE	96

13.2.1 Standards Status	96
13.2.2 wolfCOSE Status	96
13.2.3 Implemented P0 Scope	97
13.2.4 Build Selection	97
13.2.5 Path to Stable Support	97
14 Release Notes	99
14.1 wolfCOSE 2.0.0 (September 17, 2026)	99
14.1.1 Vulnerabilities	99
14.1.2 New Feature Additions	99
14.1.3 Fixes	99
14.1.4 Improvements/Optimizations	100
14.1.5 Compatibility / Migration	100
14.2 wolfCOSE 1.0.0 (June 25, 2026)	100
14.2.1 Vulnerabilities	100
14.2.2 New Feature Additions	101
14.2.3 Fixes	101
14.2.4 Improvements/Optimizations	101

1 wolfCOSE

wolfCOSE is a lightweight CBOR and COSE library for embedded systems. It implements:

- **CBOR** (RFC 8949): Concise Binary Object Representation
- **COSE** (RFC 9052/9053): CBOR Object Signing and Encryption

It uses [wolfSSL](#) as the cryptographic backend and is designed for constrained IoT devices, FIPS-bounded deployments, and anywhere you need authenticated CBOR payloads in minimal RAM.

1.1 Standards

Core specifications

- [RFC 8949](#): Concise Binary Object Representation (CBOR)
- [RFC 9052](#): CBOR Object Signing and Encryption (COSE) structures
- [RFC 9053](#): COSE initial algorithms
- [RFC 9864](#): Fully-specified algorithms for JOSE and COSE
- [RFC 9338](#): COSE countersignatures

Post-quantum cryptography

- [RFC 9964](#): ML-DSA for COSE
- [RFC 8778](#): HSS/LMS hash-based signatures for COSE

Attestation

- [RFC 9783](#): PSA attestation token profile of EAT

1.2 Key Features

Feature	Description
Complete RFC 9052	All six COSE message types — Sign1, Sign, Encrypt0, Encrypt, Mac0, Mac
Multi-signer / multi-recipient	Full COSE_Sign, COSE_Encrypt, and COSE_Mac support
Post-quantum signing	ML-DSA (FIPS 204) at all three security levels — first COSE library to ship native PQC
PSA attestation	Optional RFC 9783 EAT / PSA token verifier and issuer, including Sign1, Mac0, legacy consumption, and PSA/HSM delegated signing
Zero dynamic allocation	Heap-allocation-free, non-recursive; caller-provided buffers within a bounded, target-customizable stack ceiling — zero .data/.bss
Tiny footprint	3.5–5.1 KB COSE engine, 5.1–6.8 KB with the built-in CBOR engine (verify-only -> sign + verify); 26.2 KB -> 34.6 KB total with wolfCrypt — ES256 COSE_Sign1, dead-code-eliminated
41 algorithms	Signing, encryption, MAC, and key distribution — classical and post-quantum
FIPS 140-3 path	Sole crypto dependency is wolfCrypt FIPS Certificate #4718
CNSA 2.0 ready MISRA-C:2023	ML-DSA-44/65/87 for quantum-resistant signatures compliance striving, Single-exit pattern, no recursion, deviation-logged

1.3 Documentation

Page	Description
Getting Started	Prerequisites, building, and quick start examples
Message Types	All six RFC 9052 messages (Sign1/Sign, Encrypt0/Encrypt, Mac0/Mac) with code samples
Algorithms	Complete list of supported algorithms with COSE IDs
API Reference	Full API documentation for all functions
Macros	Configuration macros and compile-time options
Experimental	Draft feature status, supported scope, and graduation plan
PSA and EAT	RFC 9783 PSA attestation profiles, APIs, and integration guidance
Footprint	Size and speed numbers, desktop and on-device
Testing	Unit tests, coverage, and failure injection
MISRA Compliance	MISRA C:2012 and C:2023 compliance status and deviation rationale
Project Structure	Source code layout and file descriptions
STM32Cube	Install and run wolfCOSE as an STM32Cube pack on device
Release Notes	Per-version changelog and release highlights

1.4 Supported Message Types

wolfCOSE implements all six COSE message types from RFC 9052:

Message Type	Tag	Description
COSE_Sign1	18	Single signer digital signature
COSE_Sign	98	Multiple signers
COSE_Encrypt0	16	Symmetric encryption (single key)
COSE_Encrypt	96	Multi-recipient encryption
COSE_Mac0	17	Symmetric MAC (single key)
COSE_Mac	97	Multi-recipient MAC

1.5 Quick Links

- [GitHub Repository](#)
- [wolfSSL Website](#)

See [Standards](#) above for the full list of implemented RFCs.

1.6 License

wolfCOSE is free software licensed under GPLv3. For commercial licensing and support, contact [wolfSSL](#).

2 Getting Started

This guide covers prerequisites, building wolfCOSE, and basic usage examples.

2.1 Prerequisites

2.1.1 wolfSSL Installation

wolfCOSE requires wolfSSL 5.8.0 or later with the appropriate algorithms enabled. AES Key Wrap requires wolfSSL 5.9.0 or later because that release uses a constant-time integrity comparison during unwrap. Private RSA COSE_Key decoding requires wolfSSL 5.9.0 or later, and private RSA serialization requires wolfSSL 5.9.2 or later. ML-DSA requires a wolfSSL release newer than 5.9.1. HSS/LMS (RFC 8778) requires wolfSSL 5.9.2 or later, the first release whose public-key importer derives the parameter set from the key bytes.

These dependency floors are enforced at compile time whenever wolfCOSE selects the corresponding feature. With an older wolfSSL, disable unused ML-DSA or LMS support with WOLFCOSE_NO_MLDSA or WOLFCOSE_NO_LMS. Define WOLFCOSE_RSA_PUBLIC_ONLY to retain RSA-PSS and public COSE_Key support without private RSA serialization.

Here is a full-featured build using a release that meets those feature floors:

```
cd wolfssl
./autogen.sh
./configure --enable-ecc --enable-ed25519 --enable-ed448 \
            --enable-curve25519 --enable-aesgcm --enable-aesccm \
            --enable-sha384 --enable-sha512 --enable-keygen \
            --enable-rsapss --enable-chacha --enable-poly1305 \
            --enable-mldsa --enable-lms --enable-hkdf --enable-aeskeywrap
make && sudo make install
sudo ldconfig
```

2.1.2 Minimal Builds

You can enable only the algorithms you need:

ECC + AES-GCM only:

```
./configure --enable-ecc --enable-aesgcm --enable-sha384 \
            --enable-sha512 --enable-keygen
```

Post-quantum only (ML-DSA):

```
./configure --enable-mldsa --enable-sha512
```

ECDH-ES + Key Wrap (multi-recipient encryption):

```
./configure --enable-ecc --enable-aesgcm --enable-sha384 \
            --enable-sha512 --enable-keygen --enable-hkdf --enable-aeskeywrap
```

2.1.3 Feature to wolfSSL Flag Mapping

Feature	wolfSSL Configure Flags
ECC signing (ES256/384/512)	--enable-ecc --enable-keygen
EdDSA (Ed25519)	--enable-ed25519 --enable-curve25519

Feature	wolfSSL Configure Flags
EdDSA (Ed448)	--enable-ed448
AES-GCM encryption	--enable-aesgcm
AES-CCM encryption	--enable-aesccm
ChaCha20-Poly1305	--enable-chacha --enable-poly1305
ECDH-ES key agreement	--enable-ecc --enable-hkdf
AES Key Wrap	--enable-aeskeywrap (wolfSSL 5.9.0+)
Experimental COSE-HPKE	--enable-hpke --enable-ecc --enable-aesgcm
P0	--enable-keygen
RSA-PSS signing	--enable-rsapss --enable-keygen
Private RSA COSE_Key	--enable-rsapss (wolfSSL 5.9.0+)
decoding	
Private RSA COSE_Key	--enable-rsapss --enable-keygen (wolfSSL 5.9.2+)
serialization	
ML-DSA (post-quantum)	--enable-mldsa (wolfSSL newer than 5.9.1)
HSS/LMS (stateful	--enable-lms (wolfSSL 5.9.2+)
hash-based)	
AES-MAC	--enable-aescbc

2.2 Building wolfCOSE

```
git clone https://github.com/wolfSSL/wolfCOSE.git
cd wolfCOSE
make
```

2.2.1 wolfSSL Discovery

The Makefile uses pkg-config for system-installed wolfSSL when its metadata is available. This works with Homebrew installations on either supported macOS architecture:

```
brew install wolfssl pkgconf
make
```

If the package metadata is outside the default search path, set PKG_CONFIG_PATH before building:

```
PKG_CONFIG_PATH="$(brew --prefix wolfssl)/lib/pkgconfig" make
```

For a custom prefix without a .pc file, use WOLFSSL_PREFIX. For cross-builds or other custom installations, override the flags explicitly:

```
make WOLFSSL_PREFIX=/path/to/wolfssl
make WOLFSSL_CFLAGS="-isystem /path/to/wolfssl/include" \
    WOLFSSL_LIBS="-L/path/to/wolfssl/lib -lwolfssl"
```

Set WOLFSSL_PACKAGE for a non-default package name, or PKG_CONFIG to use a different metadata tool. Run make pkg-config-test to verify these discovery paths without a wolfSSL installation.

2.2.2 Build Targets

Target	Description
make all	Build libwolfcose.a (static library)
make shared	Build libwolfcose.so (shared library)
make test	Build and run CBOR and COSE unit tests
make pkg-config-test	Verify wolfSSL package discovery and overrides

Target	Description
make tool	Build CLI tool (tools/wolfcose_tool)
make tool-test	Round-trip self-test for all 17 algorithms
make demo	Build and run lifecycle demo (11 algorithms)
make demos	Build and run all basic demos
make hpke-demo	Build and run the opt-in experimental COSE-HPKE P0 demo
make c99-hpke-check	Strict C99 syntax check for all experimental HPKE paths (HPKE-enabled wolfSSL required)
make comprehensive	Build and run comprehensive algorithm tests (~240 tests)
make scenarios	Build and run real-world scenario examples
make coverage	Run tests with gcov coverage
make clean	Remove all build artifacts

2.3 Quick Start: Sign and Verify

```
#include <wolfcose/wolfcose.h>
#include <stdio.h>

int main(void)
{
    ecc_key eccKey;
    WOLFCOSE_KEY coseKey;
    uint8_t scratch[WOLFCOSE_MAX_SCRATCH_SZ];
    uint8_t out[256];
    size_t outLen;
    WC_RNG rng;

    const uint8_t payload[] = "Hello, COSE!";
    const uint8_t kid[] = "key-1";

    /* Initialize RNG */
    wc_InitRng(&rng);

    /* Generate ECC key */
    wc_ecc_init(&eccKey);
    wc_ecc_make_key(&rng, 32, &eccKey);

    /* Wrap in COSE key structure */
    wc_CoseKey_Init(&coseKey);
    wc_CoseKey_SetEcc(&coseKey, WOLFCOSE_CRV_P256, &eccKey);

    /* Sign */
    wc_CoseSign1_Sign(&coseKey, WOLFCOSE_ALG_ESP256,
        kid, sizeof(kid) - 1,
        payload, sizeof(payload) - 1,
        NULL, 0, /* no detached payload */
        NULL, 0, /* no external AAD */
        scratch, sizeof(scratch),
        out, sizeof(out), &outLen,
        &rng);

    /* Verify */
}
```

```

WOLFCOSE_HDR hdr;
const uint8_t* decoded;
size_t decodedLen;

int ret = wc_CoseSign1_Verify(&coseKey,
    out, outLen,
    NULL, 0, /* no detached payload */
    NULL, 0, /* no external AAD */
    scratch, sizeof(scratch),
    &hdr, &decoded, &decodedLen);

if (ret == WOLFCOSE_SUCCESS) {
    printf("Verified! Payload: %.*s\n", (int)decodedLen, decoded);
}

/* Cleanup */
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);

return 0;
}

```

2.4 Quick Start: Encrypt and Decrypt

```

#include <wolfcose/wolfcose.h>
#include <stdio.h>

int main(void)
{
    WOLFCOSE_KEY coseKey;
    uint8_t scratch[WOLFCOSE_MAX_SCRATCH_SZ];
    uint8_t out[256];
    uint8_t plaintext[256];
    size_t outLen, plaintextLen;

    /* 128-bit symmetric key */
    uint8_t symKey[16] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
    };

    /* 12-byte IV for AES-GCM */
    uint8_t iv[12] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
        0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b
    };

    const uint8_t data[] = "Secret message";

    /* Setup symmetric key */
    wc_CoseKey_Init(&coseKey);
    wc_CoseKey_SetSymmetric(&coseKey, symKey, sizeof(symKey));

    /* Encrypt */

```

```

wc_CoseEncrypt0_Encrypt(&coseKey, WOLFCOSE_ALG_A128GCM,
    iv, sizeof(iv),
    data, sizeof(data) - 1,
    NULL, 0, NULL, /* no detached ciphertext */
    NULL, 0, /* no external AAD */
    scratch, sizeof(scratch),
    out, sizeof(out), &outLen);

/* Decrypt */
WOLFCOSE_HDR hdr;

int ret = wc_CoseEncrypt0_Decrypt(&coseKey,
    out, outLen,
    NULL, 0, /* no detached ciphertext */
    NULL, 0, /* no external AAD */
    scratch, sizeof(scratch),
    &hdr,
    plaintext, sizeof(plaintext), &plaintextLen);

if (ret == WOLFCOSE_SUCCESS) {
    printf("Decrypted: %.*s\n", (int)plaintextLen, plaintext);
}

return 0;
}

```

2.5 Experimental COSE-HPKE P0

COSE-HPKE tracks an active Internet-Draft, so it is disabled in every build, including a normal non-lean build. It currently implements the P0 subset: HPKE base mode with DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, and AES-128-GCM. See [Configuration Macros](#) for the complete operation and compile-out gates, and [Experimental Features](#) for the draft status and graduation plan.

Build wolfSSL with HPKE support, then enable the exact send and receive paths your application needs. WOLFCOSE_EXPERIMENTAL is required with every HPKE enable macro. The standalone example supplies that acknowledgement, enables all four paths, and demonstrates both one-recipient COSE_Encrypt0 and two-recipient COSE_Encrypt key encryption:

```

cd wolfssl
./configure --enable-cryptonly --enable-hpke --enable-ecc --enable-aesgcm \
    --enable-keygen
make

cd ../wolfCOSE
make hpke-demo \
    EXTRA_CFLAGS="-I/path/to/wolfssl" \
    LDFLAGS="-L/path/to/wolfssl -lwolfssl"

```

The command-line tool is compiled with the same operation gates. `keygen -p` exports a public-only COSE_Key; keep the corresponding `-o` private key on the recipient, and use `new`, distinct non-symlink destinations for `-o` and `-p`. On POSIX builds, HPKE key generation refuses to replace an existing destination. Normalized, case-equivalent, and symlink aliases are rejected before either key is written. On non-POSIX builds, `-p` is rejected rather than weakening those key-output safeguards. The direct commands are for HPKE-0, while the `hpke-ke-*` commands use one independently HPKE-protected CEK for every recipient:

```
# Build the tool with WOLFCOSE_EXPERIMENTAL and the four
# WOLFCOSE_ENABLE_HPKE_0_* operation macros.
./tools/wolfcose_tool keygen -a HPKE-0 \
    -o recipient.private.cbor -p recipient.public.cbor
./tools/wolfcose_tool hpke0-enc -k recipient.public.cbor \
    -i config.bin -o config.hpke.cbor
./tools/wolfcose_tool hpke0-dec -k recipient.private.cbor \
    -i config.hpke.cbor -o config.out

./tools/wolfcose_tool keygen -a HPKE-0-KE \
    -o recipient-a.private.cbor -p recipient-a.public.cbor
./tools/wolfcose_tool keygen -a HPKE-0-KE \
    -o recipient-b.private.cbor -p recipient-b.public.cbor
./tools/wolfcose_tool hpke-ke-enc -a A128GCM \
    -k recipient-a.public.cbor -k recipient-b.public.cbor \
    -i config.bin -o config.multi.cbor
./tools/wolfcose_tool hpke-ke-dec -k recipient-b.private.cbor -r 1 \
    -i config.multi.cbor -o config.out
```

-r is a zero-based recipient index. The tool limits its HPKE-0-KE command paths to WOLFCOSE_TOOL_MAX_HPKE_RECIPIENTS recipient keys and indices, four by default, which can be reduced for constrained integrations. Its output buffers reserve framing for the fixed P-256 HPKE envelope, so an input at the configured WOLFCOSE_TOOL_MAX_MSG limit remains usable. HPKE base mode authenticates the recipient, not the sender. Sign or MAC the resulting message when sender authentication is required.

2.6 Quick Start: Post-Quantum Signing (ML-DSA)

```
#include <wolfcose/wolfcose.h>
#include <stdio.h>

int main(void)
{
    wc_MlDsaKey mlDsaKey;
    WOLFCOSE_KEY coseKey;
    uint8_t scratch[8192]; /* PQC needs larger scratch */
    uint8_t out[8192];
    size_t outLen;
    WC_RNG rng;

    const uint8_t payload[] = "Quantum-safe message";

    wc_InitRng(&rng);

    /* Generate ML-DSA-44 key (Level 2) */
    wc_MlDsaKey_Init(&mlDsaKey, NULL, INVALID_DEVID);
    wc_MlDsaKey_SetParams(&mlDsaKey, WC_ML_DSA_44);
    wc_MlDsaKey_MakeKey(&mlDsaKey, &rng);

    /* Wrap in COSE key. ML-DSA uses the RFC 9964 AKP key type; this is all
     * that is needed for sign/verify. To export a *private* COSE_Key, create
     * the key with wc_MlDsaKey_MakeKeyFromSeed and pass the 32-byte seed via
     * wc_CoseKey_SetMlDsa_ex (RFC 9964 private keys are the seed). */
    wc_CoseKey_Init(&coseKey);
```

```

wc_CoseKey_SetMlDsa(&coseKey, WOLFCOSE_ALG_ML_DSA_44, &mlDsaKey);

/* Sign */
wc_CoseSign1_Sign(&coseKey, WOLFCOSE_ALG_ML_DSA_44,
    NULL, 0,
    payload, sizeof(payload) - 1,
    NULL, 0, NULL, 0,
    scratch, sizeof(scratch),
    out, sizeof(out), &outLen,
    &rng);

/* Verify */
WOLFCOSE_HDR hdr;
const uint8_t* decoded;
size_t decodedLen;

wc_CoseSign1_Verify(&coseKey,
    out, outLen,
    NULL, 0, NULL, 0,
    scratch, sizeof(scratch),
    &hdr, &decoded, &decodedLen);

wc_MlDsaKey_Free(&mlDsaKey);
wc_FreeRng(&rng);

return 0;
}

```

2.7 CLI Tool

The `wolfcose_tool` provides command-line access to all wolfCOSE operations:

```

# Build the tool
make tool

# Generate keys
./tools/wolfcose_tool keygen -a ESP256 -o ec.key
./tools/wolfcose_tool keygen -a ML-DSA-44 -o pqc.key
./tools/wolfcose_tool keygen -a A128GCM -o sym.key

# Sign and verify
./tools/wolfcose_tool sign -k ec.key -a ESP256 -i data.bin -o data.cose
./tools/wolfcose_tool verify -k ec.key -i data.cose

# Encrypt and decrypt
./tools/wolfcose_tool enc -k sym.key -a A128GCM -i secret.bin -o secret.cose
./tools/wolfcose_tool dec -k sym.key -i secret.cose -o recovered.bin

# MAC operations
./tools/wolfcose_tool keygen -a HMAC256 -o hmac.key
./tools/wolfcose_tool mac -k hmac.key -a HMAC256 -i data.bin -o data.mac
./tools/wolfcose_tool macverify -k hmac.key -i data.mac

# Inspect COSE structure

```

```
./tools/wolfcose_tool info -i data.cose
```

```
# Self-test all algorithms
```

```
./tools/wolfcose_tool test --all
```

2.8 Examples Directory

The examples/ directory contains complete working examples:

File	Description
sign1_demo.c	All COSE_Sign1 algorithms
encrypt0_demo.c	All COSE_Encrypt0 algorithms
mac0_demo.c	All COSE_Mac0 algorithms
lifecycle_demo.c	Full edge-to-cloud workflow
psa_eat_demo.c	RFC 9783 device onboarding and component appraisal
psa_eat_verify_lean.c	Verify a fixed RFC 9783 token in a lean receiver

Run the complete PSA/EAT onboarding flow with:

```
make psa-eat-demo
```

The demo hashes a sample secure-partition firmware image, creates a current profile COSE_Sign1 token, selects a provisioned public IAK by UEID, proves that a wrong challenge is rejected, verifies the correct challenge, and appraises the authenticated lifecycle, device IDs, software measurement, signer ID, and version against trusted reference values.

2.8.1 Comprehensive Tests (examples/comprehensive/)

File	Description
sign_all.c	Sign1 and multi-signer matrix tests (~61 tests)
encrypt_all.c	Encrypt0 and multi-recipient matrix tests (~23 tests)
mac_all.c	Mac0 and multi-recipient matrix tests (~32 tests)
errors_all.c	Error handling and edge cases (~19 tests)

2.8.2 Real-World Scenarios (examples/scenarios/)

File	Description
firmware_update.c	Post-quantum ML-DSA firmware signing with detached payload
multi_party_approval.c	Dual-control firmware approval (ESP256 + ESP384)
iot_fleet_config.c	Encrypted config push to IoT device fleet
sensor_attestation.c	EAT-style attestation with replay protection via AAD
group_broadcast_mac.c	Authenticated broadcast to multiple subscribers

2.9 Strict Decoding (RFC 8949 Preferred Serialization)

Read this before filing an interop bug. wolfCOSE's ordinary decoder APIs accept only *deterministically encoded* CBOR. This is required by COSE (RFC 9052) and by CTAP2 canonical CBOR, but it is stricter than most general-purpose CBOR parsers, so on a device the symptom is usually “my authenticator rejects requests from client X” rather than an obvious parse bug. The optional RFC 9783 PSA/EAT verifier is a

definite-length variation-tolerant exception; that profile behavior is not exposed through the ordinary COSE API.

The following two rules apply to `wc_CBOR_Decode*`(), `wc_CoseKey_Decode*`(), and the ordinary `_Verify` / `_Decrypt` functions:

Rule	Example rejected input	Error
Arguments must use the shortest additional-information form (RFC 8949 Section 4.2.1)	0x18 0x17 for 23 (must be 0x17); 0x19 0x00 0x64 for 100 (must be 0x18 0x64)	WOLFCOSE_E_CBOR_MALFORMED
Indefinite lengths are not accepted (additional information 31)	0x5F ... 0xFF (chunked bstr), 0x9F ... 0xFF (open array)	WOLFCOSE_E_UNSUPPORTED

Related strictness that surprises integrators for the same reason:

- Trailing bytes after the encoded object are rejected. `inSz` must be exactly the object length, not the capacity of the buffer holding it. Use `wc_CBOR_SkipItem()` to carve out the exact byte range of an embedded item.
- Two-byte simple values below 32 are malformed, per RFC 8949.
- All encountered CBOR text strings are validated as UTF-8. This also applies to `wc_CBOR_DecodeHead()`, `wc_CBOR_DecodeLabel()`, `wc_CBOR_Skip()`, and `wc_CBOR_SkipItem()` when they only inspect or traverse the string; invalid text returns `WOLFCOSE_E_CBOR_MALFORMED`.
- EC2 coordinates must be exactly the curve size, with leading zeros preserved (RFC 9053 Section 7.1.1) - a 31-byte P-256 x is rejected, not left-padded.
- A duplicate label in a header or COSE_Key map is rejected.
- COSE header maps accept unknown text labels and reject their duplicates. Registered COSE parameters remain numeric; "alg" is not an alias for integer label 1. COSE_Key maps always require integer labels.

The strict decoding rules are not configurable for the ordinary API: relaxing them would let a signature or MAC be recomputed over a re-encoding of the same data. The text-label option expands accepted label types without relaxing CBOR encoding rules. The optional PSA/EAT verifier's variation-tolerant path still rejects indefinite CBOR.

2.10 Cross-Compilation

For embedded targets:

```
make CC=arm-none-eabi-gcc \
    CFLAGS="-std=c99 -Os -mcpu=cortex-m4 -mthumb \
            -I./include -I/path/to/wolfssl/include \
            -DWOLFSSL_USER_SETTINGS"
```

Provide a `user_settings.h` with your wolfSSL configuration instead of `wolfssl/options.h`.

2.11 Stack Budget

Per-function stack usage (from `-fstack-usage`, GCC, `-Os`, `aarch64`):

Function	Stack (bytes)
<code>wc_CoseSign1_Sign</code>	464
<code>wc_CoseSign1_Verify</code>	288
<code>wc_CoseEncrypt0_Encrypt</code>	1120
<code>wc_CoseEncrypt0_Decrypt</code>	1072
<code>wc_CoseMac0_Create</code>	1104
<code>wc_CoseMac0_Verify</code>	1072
<code>wc_CoseKey_Encode</code>	352
<code>wc_CoseKey_Decode</code>	224
<code>wc_CBOR_Skip</code>	112
CBOR encode/decode	0-48

2.12 Next Steps

- **Algorithms:** See all supported algorithms
- **API Reference:** Complete function documentation
- **Macros:** Configure compile-time options
- **Testing:** Run tests and measure coverage

3 COSE Message Types

wolfCOSE implements the complete RFC 9052 message set — all six COSE structures, including the multi-actor variants (COSE_Sign, COSE_Encrypt, COSE_Mac) that most embedded C COSE libraries omit.

Message	RFC 9052	Tag	API
COSE_Sign1	Sec. 4.2	18	wc_CoseSign1_Sign / wc_CoseSign1_Verify
COSE_Sign	Sec. 4.1	98	wc_CoseSign_Sign / wc_CoseSign_Verify
COSE_Encrypt0	Sec. 5.2	16	wc_CoseEncrypt0_Encrypt / wc_CoseEncrypt0_Decrypt
COSE_Encrypt	Sec. 5.1	96	wc_CoseEncrypt_Encrypt / wc_CoseEncrypt_Decrypt
COSE_Mac0	Sec. 6.2	17	wc_CoseMac0_Create / wc_CoseMac0_Verify
COSE_Mac	Sec. 6.1	97	wc_CoseMac_Create / wc_CoseMac_Verify
COSE_Key / COSE_KeySet	Sec. 7	(none)	wc_CoseKey_Encode / wc_CoseKey_Decode

Every message can be built attached or detached, with optional external AAD, and every algorithm in [Algorithms](#) is available to every message type that accepts it.

3.1 COSE_Sign1 — single-signer signature (RFC 9052 Sec. 4.2)

COSE_Sign1 is the most common COSE message: one signer, one signature.

```
WOLFCOSE_KEY key;
uint8_t out[1024];
size_t outLen;
WC_RNG rng;

wc_InitRng(&rng);
wc_CoseKey_Init(&key);
wc_CoseKey_SetEcc(&key, WOLFCOSE_CRV_P256, &eccPriv);

int ret = wc_CoseSign1_Sign(&key, WOLFCOSE_ALG_ESP256,
                           payload, payloadLen,
                           NULL, 0, /* no detached payload */
                           NULL, 0, /* no external AAD */
                           scratch, sizeof(scratch),
                           out, sizeof(out), &outLen,
                           &rng);
```

3.2 COSE_Sign — multi-signer signature (RFC 9052 Sec. 4.1)

COSE_Sign carries N independent signatures over the same payload. Use it for dual-control approval, certificate chains where multiple parties attest, or hybrid classical/PQC signatures during migration. Each signer has its own algorithm and kid.

```
WOLFCOSE_SIGNATURE signers[2] = {
    { .algId = WOLFCOSE_ALG_ESP256,
      .key   = &vendorKey,
      .kid   = (const uint8_t*)"vendor-2026", .kidLen = 11 },
    { .algId = WOLFCOSE_ALG_ML_DSA_65, /* hybrid: classical + PQC */
      .key   = &oemKey,
      .kid   = (const uint8_t*)"oem-pqc-1", .kidLen = 9 },
};
```

```
ret = wc_CoseSign_Sign(signers, 2,
                      firmware, firmwareLen,
                      NULL, 0, /* attached payload */
                      NULL, 0, /* no external AAD */
                      scratch, sizeof(scratch),
                      out, sizeof(out), &outLen,
                      &rng);
```

Verifiers select which signature to check by index:

```
ret = wc_CoseSign_Verify(&vendorPubKey, /*signerIndex=*/0,
                       in, inLen,
                       NULL, 0, NULL, 0,
                       scratch, sizeof(scratch),
                       &hdr, &payload, &payloadLen);
```

3.3 RFC 9338 countersignatures

A countersignature lets a second party sign an existing COSE cryptographic object. The original signer and countersigner remain independently verifiable. For example, a vendor can sign firmware and a release authority can countersign the resulting COSE_Sign1 after policy checks, without replacing the vendor signature or copying the firmware into a new envelope.

wolfCOSE can add countersignatures to tagged COSE_Sign1, COSE_Sign, COSE_Encrypt0, COSE_Encrypt, COSE_Mac0, and COSE_Mac messages. Countersigners use ECDSA, EdDSA, RSA-PSS, or ML-DSA; HSS/LMS is rejected with WOLFCOSE_E_COSE_BAD_ALG, and an HSS/LMS-only build does not enable the feature.

```
WOLFCOSE_COUNTERSIGNATURE approval = {
    .algId = WOLFCOSE_ALG_ESP256,
    .key    = &releaseKey,
    .kid    = (const uint8_t*)"release-2026",
    .kidLen = 12
};
```

```
ret = wc_Cose_AddCounterSignature(&approval,
                                signedMsg, signedMsgLen,
                                NULL, 0, /* detached payload */
                                aad, aadLen,
                                scratch, sizeof(scratch),
                                approved, sizeof(approved), &approvedLen,
                                &rng);
```

```
ret = wc_Cose_VerifyCounterSignature(&releasePubKey, 0,
                                    approved, approvedLen,
                                    NULL, 0, aad, aadLen,
                                    scratch, sizeof(scratch),
                                    &counterHdr);
```

Verify the original COSE operation separately. For a COSE_Sign1, call wc_CoseSign1_Verify() as well as wc_Cose_VerifyCounterSignature(). Appending another full countersignature preserves existing values; select one with the zero-based counterIndex.

wc_Cose_AddCounterSignature0() emits the abbreviated form when the algorithm and key selection are known through trusted application context. Exact in-place growth is supported by passing the

same buffer for in and out, provided the buffer has enough capacity. Detached payloads or ciphertext and external AAD must be supplied again when adding or verifying the countersignature.

Creation emits the RFC 9338 V2 labels 11 and 12. Verification also accepts the legacy RFC 8152 labels 7 and 9 for deployed interoperability.

The command-line tool can countersign an existing message and verify a selected full countersignature:

```
wolfcose_tool countersign -k release-key.cbor -a ESP256 \
-i signed.cose -o approved.cose
wolfcose_tool counterverify -k release-public.cbor \
-i approved.cose --index 0
```

3.3.1 Relationship to SCITT

Countersignatures are useful for supply-chain endorsement, notary, approval, and separation-of-duty workflows. They can add an independently verifiable approval layer to a signed statement.

They do not by themselves implement a SCITT transparency service. The SCITT architecture is standardized in [RFC 9943](#), and its receipts use the formats in [RFC 9942](#). A complete SCITT product also needs statement registration and policy, receipt generation and validation, and transparency-service operation. wolfCOSE countersignatures provide a complementary COSE building block for deployments that need an extra independent endorsement.

3.4 COSE_Encrypt0 — single-recipient AEAD (RFC 9052 Sec. 5.2)

Direct symmetric AEAD. The caller already has the content encryption key (CEK).

```
ret = wc_CoseEncrypt0_Encrypt(&aesKey, WOLFCOSE_ALG_A128GCM,
                              iv, sizeof(iv),
                              plaintext, plaintextLen,
                              NULL, 0, NULL, 0,
                              scratch, sizeof(scratch),
                              out, sizeof(out), &outLen,
                              &rng);
```

3.5 COSE_Encrypt — multi-recipient AEAD (RFC 9052 Sec. 5.1)

COSE_Encrypt produces one ciphertext addressable by any of N recipients. Each recipient entry describes how that recipient learns the CEK: pre-shared (Direct), wrapped under a recipient KEK (A128KW / A192KW / A256KW), or derived via ECDH.

```
WOLFCOSE_RECIPIENT recipients[3] = {
    { .algId = WOLFCOSE_ALG_DIRECT,
      .key   = &fleetSharedKey,
      .kid   = (const uint8_t*)"fleet-2026", .kidLen = 10 },
    { .algId = WOLFCOSE_ALG_A256KW,
      .key   = &device42Kek,
      .kid   = (const uint8_t*)"dev-42",      .kidLen = 6 },
    { .algId = WOLFCOSE_ALG_ECDH_ES_HKDF_256,
      .key   = &device43Pub,
      .kid   = (const uint8_t*)"dev-43",      .kidLen = 6 },
};
```

```
ret = wc_CoseEncrypt_Encrypt(recipients, 3,
```

```

WOLFCOSE_ALG_A128GCM,          /* content alg */
iv, sizeof(iv),
config, configLen,
NULL, 0, NULL, 0,
scratch, sizeof(scratch),
out, sizeof(out), &outLen,
&rng);

```

A receiving device decrypts by selecting its own recipient index:

```

ret = wc_CoseEncrypt_Decrypt(&myRecipient, /*recipientIndex=*/1,
                             in, inLen,
                             NULL, 0, NULL, 0,
                             scratch, sizeof(scratch),
                             &hdr, plaintext, sizeof(plaintext), &plaintextLen);

```

3.6 COSE_Mac0 — single-recipient MAC (RFC 9052 Sec. 6.2)

Direct MAC over a payload using a pre-shared key.

```

ret = wc_CoseMac0_Create(&hmacKey, WOLFCOSE_ALG_HMAC_256_256,
                        payload, payloadLen,
                        NULL, 0, NULL, 0,
                        scratch, sizeof(scratch),
                        out, sizeof(out), &outLen);

```

3.7 COSE_Mac — multi-recipient MAC (RFC 9052 Sec. 6.1)

COSE_Mac carries one MAC tag plus per-recipient envelopes describing how each recipient learns the MAC key. Useful for group broadcast scenarios where every subscriber needs to authenticate the same payload but holds a different KEK.

```
WOLFCOSE_RECIPIENT subscribers[N] = { /* one entry per group member */ };
```

```

ret = wc_CoseMac_Create(subscribers, N,
                        WOLFCOSE_ALG_HMAC_256_256,
                        payload, payloadLen,
                        NULL, 0, NULL, 0,
                        scratch, sizeof(scratch),
                        out, sizeof(out), &outLen);

```

3.8 Detached payloads

Every Sign*, Encrypt*, and Mac* API accepts a detached payload — the COSE message stores only the signature/MAC/ciphertext and a nil payload field, while the actual bytes live elsewhere. This is essential when signing large objects (firmware, OTA images) where you don't want to copy the payload through the COSE buffer. Pass the payload via the detachedPayload / detachedLen arguments instead of payload / payloadLen, and the same on verify.

3.9 Compile-time stripping

You only pay for the message types you use. Strip whole message types with the matching WOLF-COSE_NO_* macros — each is independent:

```
-DWOLFCOSE_NO_SIGN -DWOLFCOSE_NO_ENCRYPT -DWOLFCOSE_NO_MAC    /* drop all
↳ multi-actor types */
-DWOLFCOSE_NO_MAC0 -DWOLFCOSE_NO_MAC                          /* drop all MAC */
```

For common minimal builds, use a build profile instead of hand-listing macros — these are the supported, CI-tested configurations (see [Macros](#) -> Build Profiles):

Profile	Result
WOLFCOSE_LEAN	single-actor only: COSE_Sign1 / Encrypt0 / Mac0
WOLFCOSE_LEAN_VERIFY	COSE_Sign1 verify-only (no signing, no RNG)
WOLFCOSE_LEAN_MLDSA	ML-DSA COSE_Sign1 sign + verify
WOLFCOSE_LEAN_VERIFY_MLDSA	ML-DSA COSE_Sign1 verify-only

A minimal Sign1-verify-only build (WOLFCOSE_LEAN_VERIFY) is about **5.1 KB** of wolfCOSE library code (3.5 KB COSE engine + the built-in CBOR engine), or **26.2 KB** total flash with a minimal wolfCrypt P-256 ECDSA/SHA-256 backend — rising to **6.8 KB / 34.6 KB** for sign + verify.

3.10 See also

- [Algorithms](#) — full algorithm list with COSE IDs and wolfCrypt guards
- [API Reference](#) — function signatures, structures, and error codes
- [Macros](#) — every WOLFCOSE_* and WOLFCOSE_NO_* compile-time toggle
- [examples/scenarios/](#) — `multi_party_approval.c`, `iot_fleet_config.c`, `group_broadcast_mac.c` show real multi-actor flows

4 Supported Algorithms

wolfCOSE supports 46 algorithms across signing, encryption, MAC, and key distribution. This page provides the complete list with COSE algorithm IDs and required wolfSSL compile-time guards. All algorithms are usable in both single-actor messages (Sign1/Encrypt0/Mac0) and multi-actor messages (Sign/Encrypt/Mac). See [Message Types](#) for details.

4.1 COSE_Sign1 (Digital Signatures)

Algorithm	COSE ID	wolfCrypt Guard	Notes
ESP256	-9	HAVE_ECC	ECDSA with P-256 / SHA-256 (RFC 9864)
ESP384	-51	HAVE_ECC	ECDSA with P-384 / SHA-384 (RFC 9864)
ESP512	-52	HAVE_ECC	ECDSA with P-521 / SHA-512 (RFC 9864)
Ed25519	-19	HAVE_ED25519	EdDSA on Curve25519 (RFC 9864)
Ed448	-53	HAVE_ED448	EdDSA on Curve448 (Goldilocks) (RFC 9864)
ES256	-7	HAVE_ECC	Deprecated by RFC 9864; needs WOLF - COSE_ENABLE_DEPRECATED_AL
ES384	-35	HAVE_ECC	Deprecated by RFC 9864; needs WOLF - COSE_ENABLE_DEPRECATED_AL
ES512	-36	HAVE_ECC	Deprecated by RFC 9864; needs WOLF - COSE_ENABLE_DEPRECATED_AL
EdDSA	-8	HAVE_ED25519 / HAVE_ED448	Deprecated by RFC 9864 (curve taken from the key); needs WOLF - COSE_ENABLE_DEPRECATED_AL
PS256	-37	WC_RSA_PSS	RSA-PSS with SHA-256
PS384	-38	WC_RSA_PSS	RSA-PSS with SHA-384

Algorithm	COSE ID	wolfCrypt Guard	Notes
PS512	-39	WC_RSA_PSS	RSA-PSS with SHA-512
ML-DSA-44	-48	WOLFSSL_HAVE_MLDSA	Post-quantum (FIPS 204)
ML-DSA-65	-49	WOLFSSL_HAVE_MLDSA	Post-quantum (FIPS 204)
ML-DSA-87	-50	WOLFSSL_HAVE_MLDSA	Post-quantum (FIPS 204)
HSS-LMS	-46	WOLFSSL_HAVE_LMS	Post-quantum stateful hash-based (RFC 8778, SP 800-208); COSE_Key kty 5; signature size follows the key's parameter set; signing state is caller-managed, verify-only builds pair with WOLFSSL_LMS_VERIFY_ONLY

4.1.1 Migrating from RFC 9053 Signature IDs

RFC 9864 deprecates the polymorphic RFC 9053 signature IDs. Default wolfCOSE builds accept the fully specified replacements and reject the deprecated IDs with WOLF_COSE_E_COSE_BAD_ALG.

Deprecated RFC 9053 ID	RFC 9864 replacement
ES256 (-7)	ESP256 (-9)
ES384 (-35)	ESP384 (-51)
ES512 (-36)	ESP512 (-52)
EdDSA (-8) with an Ed25519 key	Ed25519 (-19)
EdDSA (-8) with an Ed448 key	Ed448 (-53)

Use the replacement ID when creating new messages. The protected alg value is part of the signed Sig_structure, so an existing message must be signed again rather than relabeled. Define WOLF_COSE_ENABLE_DEPRECATED_ALGS when compatibility with peers that still emit the RFC 9053 IDs is required. See [Deprecated Algorithm IDs](#) for configuration details.

4.1.2 ML-DSA Signature Sizes

Algorithm	Signature Size	Public Key Size
ML-DSA-44	2,420 bytes	1,312 bytes
ML-DSA-65	3,293 bytes	1,952 bytes
ML-DSA-87	4,595 bytes	2,592 bytes

4.2 COSE_Encrypt0 (Authenticated Encryption)

Algorithm	COSE ID	wolfCrypt Guard	Notes
A128GCM	1	HAVE_AESGCM	AES-GCM 128-bit
A192GCM	2	HAVE_AESGCM	AES-GCM 192-bit
A256GCM	3	HAVE_AESGCM	AES-GCM 256-bit
ChaCha20/Poly1305	24	HAVE_CHACHA && HAVE_POLY1305	256-bit, software- friendly
AES-CCM-16-64-128	10	HAVE_AESCCM	128-bit key, 8-byte tag
AES-CCM-16-64-256	11	HAVE_AESCCM	256-bit key, 8-byte tag
AES-CCM-64-64-128	12	HAVE_AESCCM	128-bit key, 8-byte tag, short nonce
AES-CCM-64-64-256	13	HAVE_AESCCM	256-bit key, 8-byte tag, short nonce
AES-CCM-16-128-128	30	HAVE_AESCCM	128-bit key, 16-byte tag
AES-CCM-16-128-256	31	HAVE_AESCCM	256-bit key, 16-byte tag
AES-CCM-64-128-128	32	HAVE_AESCCM	128-bit key, 16-byte tag, short nonce
AES-CCM-64-128-256	33	HAVE_AESCCM	256-bit key, 16-byte tag, short nonce

4.2.1 AES-CCM Naming Convention

The AES-CCM algorithm names follow the pattern AES-CCM- {L} - {T} - {K}: - **L**: Length field size (16 or 64 bits for nonce) - **T**: Tag size (64 or 128 bits) - **K**: Key size (128 or 256 bits)

4.3 COSE_Mac0 (Message Authentication)

Algorithm	COSE ID	wolfCrypt Guard	Notes
HMAC 256/256	5	!NO_HMAC	SHA-256, 32-byte tag
HMAC 384/384	6	WOLFSSL_SHA384	SHA-384, 48-byte tag
HMAC 512/512	7	WOLFSSL_SHA512	SHA-512, 64-byte tag

Algorithm	COSE ID	wolfCrypt Guard	Notes
AES-MAC-128/64	14	HAVE_AES_CBC	128-bit key, 8-byte tag
AES-MAC-256/64	15	HAVE_AES_CBC	256-bit key, 8-byte tag
AES-MAC-128/128	25	HAVE_AES_CBC	128-bit key, 16-byte tag
AES-MAC-256/128	26	HAVE_AES_CBC	256-bit key, 16-byte tag

4.4 Key Distribution (Multi-Recipient)

Used with COSE_Encrypt and COSE_Mac for multi-recipient messages:

Algorithm	COSE ID	wolfCrypt Guard	Notes
Direct	-6	always	Pre-shared symmetric key
A128KW	-3	HAVE_AES_KEYWRAP	AES Key Wrap 128-bit
A192KW	-4	HAVE_AES_KEYWRAP	AES Key Wrap 192-bit
A256KW	-5	HAVE_AES_KEYWRAP	AES Key Wrap 256-bit
ECDH-ES+HKDF-256	-25	HAVE_ECC && HAVE_HKDF	Ephemeral-Static ECDH
ECDH-ES+HKDF-512	-26	HAVE_ECC && HAVE_HKDF	Ephemeral-Static ECDH
ECDH-SS+HKDF-256	-27	HAVE_ECC && HAVE_HKDF	Static-Static ECDH
ECDH-SS+HKDF-512	-28	HAVE_ECC && HAVE_HKDF	Static-Static ECDH
ECDH-ES+A128KW	-29	HAVE_ECC && HAVE_HKDF && HAVE_AES_KEYWRAP	ECDH + Key Wrap
ECDH-ES+A192KW	-30	HAVE_ECC && HAVE_HKDF && HAVE_AES_KEYWRAP	ECDH + Key Wrap
ECDH-ES+A256KW	-31	HAVE_ECC && HAVE_HKDF && HAVE_AES_KEYWRAP	ECDH + Key Wrap

AES Key Wrap-based algorithms also require wolfSSL 5.9.0 or later.

4.5 Key Types

COSE kty	Value	Guard	Algorithms
OKP	1	HAVE_ED25519 / HAVE_ED448	Ed25519, Ed448 (EdDSA)
EC2	2	HAVE_ECC	ESP256, ESP384, ESP512 (ES256, ES384, ES512)
RSA	3	WC_RSA_PSS	PS256, PS384, PS512
Symmetric	4	always	AES-GCM, AES-CCM, ChaCha20, HMAC
AKP	7	WOLFSSL_HAVE_MLDSA	ML-DSA (RFC 9964)
HSS-LMS	5	WOLFSSL_HAVE_LMS	HSS/LMS (RFC 8778)

ML-DSA keys use the RFC 9964 **AKP** (Algorithm Key Pair) key type: the `alg` parameter is REQUIRED, the public key is in `pub` (-1), and the private key is the 32-byte seed in `priv` (-2). There is no `crv` parameter.

4.6 Curves

COSE crv	Value	Description
P-256	1	NIST P-256 (secp256r1)
P-384	2	NIST P-384 (secp384r1)
P-521	3	NIST P-521 (secp521r1)
Ed25519	6	Ed25519 for signatures
Ed448	7	Ed448 for signatures

4.7 COSE Tags

Tag	Value	Message Type
COSE_Encrypt0	16	Symmetric encryption (single key)
COSE_Mac0	17	Symmetric MAC (single key)
COSE_Sign1	18	Single signer signature
COSE_Encrypt	96	Multi-recipient encryption
COSE_Mac	97	Multi-recipient MAC
COSE_Sign	98	Multi-signer signature

4.8 Algorithm Constants in Code

wolfCOSE defines these constants in `wolfcose.h`:

```

/* Signature algorithms (RFC 9864 fully-specified) */
#define WOLFCOSE_ALG_ESP256      (-9)
#define WOLFCOSE_ALG_ESP384      (-51)
#define WOLFCOSE_ALG_ESP512      (-52)
#define WOLFCOSE_ALG_ED25519     (-19)
#define WOLFCOSE_ALG_ED448       (-53)
/* These four IDs are deprecated by RFC 9864 and require
 * WOLFCOSE_ENABLE_DEPRECATED_ALGS. */
#define WOLFCOSE_ALG_ES256       (-7)
#define WOLFCOSE_ALG_ES384       (-35)
#define WOLFCOSE_ALG_ES512       (-36)
#define WOLFCOSE_ALG_EDDSA       (-8)
/* RSA-PSS IDs remain enabled by default. */
#define WOLFCOSE_ALG_PS256       (-37)
#define WOLFCOSE_ALG_PS384       (-38)
#define WOLFCOSE_ALG_PS512       (-39)
#define WOLFCOSE_ALG_ML_DSA_44   (-48)
#define WOLFCOSE_ALG_ML_DSA_65   (-49)
#define WOLFCOSE_ALG_ML_DSA_87   (-50)
#define WOLFCOSE_ALG_HSS_LMS     (-46)

/* Encryption algorithms */
#define WOLFCOSE_ALG_A128GCM      (1)
#define WOLFCOSE_ALG_A192GCM      (2)
#define WOLFCOSE_ALG_A256GCM      (3)
#define WOLFCOSE_ALG_CHACHA20_POLY1305 (24)
#define WOLFCOSE_ALG_AES_CCM_16_64_128 (10)
#define WOLFCOSE_ALG_AES_CCM_16_64_256 (11)

```

```

/* ... and more */

/* MAC algorithms */
#define WOLFCOSE_ALG_HMAC_256_256    (5)
#define WOLFCOSE_ALG_HMAC_384_384    (6)
#define WOLFCOSE_ALG_HMAC_512_512    (7)
#define WOLFCOSE_ALG_AES_MAC_128_64  (14)
/* ... and more */

/* Key distribution */
#define WOLFCOSE_ALG_DIRECT            (-6)
#define WOLFCOSE_ALG_A128KW           (-3)
#define WOLFCOSE_ALG_A192KW           (-4)
#define WOLFCOSE_ALG_A256KW           (-5)
#define WOLFCOSE_ALG_DIRECT_HKDF_SHA_256 (-10)
#define WOLFCOSE_ALG_DIRECT_HKDF_SHA_512 (-11)
#define WOLFCOSE_ALG_DIRECT_HKDF_AES_128 (-12)
#define WOLFCOSE_ALG_DIRECT_HKDF_AES_256 (-13)
#define WOLFCOSE_ALG_ECDH_ES_HKDF_256  (-25)
#define WOLFCOSE_ALG_ECDH_ES_HKDF_512  (-26)

/* Key types */
#define WOLFCOSE_KTY_OKP              (1)
#define WOLFCOSE_KTY_EC2              (2)
#define WOLFCOSE_KTY_SYMMETRIC        (4)

/* Curves */
#define WOLFCOSE_CRV_P256             (1)
#define WOLFCOSE_CRV_P384             (2)
#define WOLFCOSE_CRV_P521             (3)
#define WOLFCOSE_CRV_ED25519          (6)
#define WOLFCOSE_CRV_ED448            (7)

```

4.9 Roadmap

Future algorithm support planned:

Algorithm	Standard	Description
ML-KEM	FIPS 203 (Kyber)	Post-quantum key encapsulation for COSE_Encrypt (IETF draft, no codepoints yet)
ESB256/320/384/512	RFC 9864	ECDSA on the Brainpool curves (Recommended: No); needs wolfSSL HAVE_ECC_BRAINPOOL
XMSS	NIST SP 800-208	Hash-based stateful signatures (no COSE codepoints assigned yet)
SLH-DSA	SPHINCS+	Stateless hash-based signatures

4.10 See Also

- **Getting Started:** Build instructions and examples
- **API Reference:** Function documentation
- **Macros:** Compile-time configuration

5 API Reference

Complete API documentation for wolfCOSE (RFC 9052/9053 COSE and RFC 9338 countersignature implementation).

5.1 Table of Contents

- Data Structures
 - COSE_Key API
 - COSE_Sign1 API
 - COSE Countersignature API
 - COSE_Encrypt0 API
 - COSE_Mac0 API
 - COSE_Sign API (Multi-Signer)
 - COSE_Encrypt API (Multi-Recipient)
 - COSE_Mac API (Multi-Recipient)
 - PSA/EAT API
 - CBOR API
 - Error Codes
-

5.2 Data Structures

5.2.1 WOLFCOSE_KEY

```
typedef struct WOLFCOSE_KEY {
    int32_t kty;           /* Key type: WOLFCOSE_KTY_EC2, WOLFCOSE_KTY_OKP, etc. */
    int32_t alg;           /* Algorithm hint (optional) */
    int32_t crv;           /* Curve for EC2/OKP keys */
    union {
        ecc_key* ecc;
        ed25519_key* ed25519;
        ed448_key* ed448;
        RsaKey* rsa;
        wc_MlDsaKey* mldsa;
        struct {
            const uint8_t* key;
            size_t keyLen;
        } symm;
    } key;
} WOLFCOSE_KEY;
```

Pointer-based key structure (~48 bytes). Caller owns underlying wolfCrypt keys.

5.2.2 WOLFCOSE_HDR

```
typedef struct WOLFCOSE_HDR {
    int32_t alg;           /* Algorithm from either header bucket */
    const uint8_t* kid;    /* Key ID (zero-copy pointer) */
    size_t kidLen;
    const uint8_t* iv;     /* IV from unprotected header */
    size_t ivLen;
```

```

    const uint8_t* partialIv; /* Partial IV from unprotected header */
    size_t partialIvLen;
    int32_t contentType;      /* Content type from either header bucket */
    uint8_t flags;            /* WOLFCOSE_HDR_FLAG_* */
} WOLFCOSE_HDR;

```

Parsed COSE header information. `alg` and `contentType` may come from either the protected or unprotected header bucket; `kid`, `iv`, and `partialIv` are unprotected metadata. `WOLFCOSE_HDR_FLAG_CONTENT_TYPE_UNPROTECTED` is set when the unprotected bucket contains the content-type label. Integer values are stored in `contentType`; text-string values are accepted but not retained. `wolfCOSE` pins an unprotected algorithm to the supplied key where cryptographic policy requires it, but applications must not treat the other returned fields as authenticated policy unless they independently pin them.

All message APIs require the external AAD pointer to be non-NULL when its length is non-zero.

5.2.3 WOLFCOSE_SIGNATURE

```

typedef struct WOLFCOSE_SIGNATURE {
    int32_t algId;          /* Signature algorithm */
    WOLFCOSE_KEY* key;      /* Signing key */
    const uint8_t* kid;     /* Key identifier */
    size_t kidLen;
} WOLFCOSE_SIGNATURE;

```

Signer information for COSE_Sign multi-signer messages.

5.2.4 WOLFCOSE_COUNTERSIGNATURE

```

typedef struct WOLFCOSE_COUNTERSIGNATURE {
    int32_t algId;
    WOLFCOSE_KEY* key;
    const uint8_t* kid;
    size_t kidLen;
} WOLFCOSE_COUNTERSIGNATURE;

```

```

typedef struct WOLFCOSE_COUNTERSIGNATURE0 {
    int32_t algId;
    WOLFCOSE_KEY* key;
} WOLFCOSE_COUNTERSIGNATURE0;

```

The full form carries protected algorithm metadata and an optional key ID. The abbreviated form carries only signature bytes, so its algorithm and key selection must be supplied by trusted application context.

5.2.5 WOLFCOSE_RECIPIENT

```

typedef struct WOLFCOSE_RECIPIENT {
    int32_t algId;          /* Key distribution algorithm */
    WOLFCOSE_KEY* key;      /* Recipient key */
    const uint8_t* kid;     /* Key identifier */
} WOLFCOSE_RECIPIENT;

```

```
    size_t kidLen;
} WOLFCOSE_RECIPIENT;
```

Recipient information for COSE_Encrypt and COSE_Mac multi-recipient messages.

5.2.6 WOLFCOSE_CBOR_CTX

```
typedef struct WOLFCOSE_CBOR_CTX {
    uint8_t*    buf;           /* Encode output */
    const uint8_t* cbuf;       /* Decode input */
    size_t      bufSz;         /* Buffer size */
    size_t      idx;           /* Current position */
} WOLFCOSE_CBOR_CTX;
```

CBOR encoder/decoder context. Use the initializer appropriate to its mode; the initializers clear the opposite pointer so encode operations cannot write through decoder input. Variation-tolerant parsing is private to the optional PSA/EAT verifier and is not stored in caller-owned context state.

5.3 COSE_Key API

5.3.1 wc_CoseKey_Init

```
int wc_CoseKey_Init(WOLFCOSE_KEY* key);
```

Initialize a COSE key structure.

Parameters: | Name | Description | |---|-----| | key | Pointer to COSE key structure to initialize |

Returns: WOLFCOSE_SUCCESS (0) or WOLFCOSE_E_INVALID_ARG

5.3.2 wc_CoseKey_Free

```
void wc_CoseKey_Free(WOLFCOSE_KEY* key);
```

Free a COSE key structure. Does NOT free the underlying wolfCrypt key (caller owns key lifecycle).

Parameters: | Name | Description | |---|-----| | key | Pointer to COSE key structure to free |

5.3.3 wc_CoseKey_SetEcc

```
int wc_CoseKey_SetEcc(WOLFCOSE_KEY* key, int32_t crv, ecc_key* eccKey);
```

Associate an ECC key with a COSE key structure.

Parameters: | Name | Description | |---|-----| | key | Pointer to initialized COSE key | | crv | Curve identifier: WOLFCOSE_CRV_P256, WOLFCOSE_CRV_P384, or WOLFCOSE_CRV_P521 | | eccKey | Pointer to initialized wolfCrypt ECC key (caller-owned) |

Returns: WOLFCOSE_SUCCESS or error code

5.3.4 wc_CoseKey_SetEd25519

```
int wc_CoseKey_SetEd25519(WOLFCOSE_KEY* key, ed25519_key* edKey);
```

Associate an Ed25519 key with a COSE key structure.

Parameters: | Name | Description | |——|————| | key | Pointer to initialized COSE key | | edKey | Pointer to initialized wolfCrypt Ed25519 key (caller-owned) |

Returns: WOLFCOSE_SUCCESS or error code

5.3.5 wc_CoseKey_SetEd448

```
int wc_CoseKey_SetEd448(WOLFCOSE_KEY* key, ed448_key* edKey);
```

Associate an Ed448 key with a COSE key structure.

Parameters: | Name | Description | |——|————| | key | Pointer to initialized COSE key | | edKey | Pointer to initialized wolfCrypt Ed448 key (caller-owned) |

Returns: WOLFCOSE_SUCCESS or error code

Requires: HAVE_ED448

5.3.6 wc_CoseKey_SetMLDsa

```
int wc_CoseKey_SetMLDsa(WOLFCOSE_KEY* key, int32_t alg, wc_MLDsaKey* mLDsaKey);
```

Associate an ML-DSA (FIPS 204) post-quantum key with a COSE key structure. The key is encoded as an RFC 9964 **AKP** COSE_Key (kty = 7, REQUIRED alg, public key in pub (-1)). Use this for public-key, sign, and verify use. To encode a *private* COSE_Key (whose private value is the 32-byte seed), use wc_CoseKey_SetMLDsa_ex and supply the seed.

Parameters: | Name | Description | |——|————| | key | Pointer to initialized COSE key | | alg | Algorithm: WOLFCOSE_ALG_ML_DSA_44, WOLFCOSE_ALG_ML_DSA_65, or WOLFCOSE_ALG_ML_DSA_87 | | mLDsaKey | Pointer to initialized wolfCrypt ML-DSA key (caller-owned) |

Returns: WOLFCOSE_SUCCESS or error code

Requires: WOLFSSL_HAVE_MLDSA

5.3.7 wc_CoseKey_SetMLDsa_ex

```
int wc_CoseKey_SetMLDsa_ex(WOLFCOSE_KEY* key, int32_t alg,
                           wc_MLDsaKey* mLDsaKey,
                           const uint8_t* seed, size_t seedLen);
```

Like wc_CoseKey_SetMLDsa, but also attaches the 32-byte ML-DSA seed used to create the key. RFC 9964 represents an ML-DSA private key as the seed, and wolfCrypt does not retain it, so the caller (who created the key via wc_MLDsaKey_MakeKeyFromSeed) must supply it here to encode a private COSE_Key.

Parameters: | Name | Description | |——|————| | key | Pointer to initialized COSE key | | alg | Algorithm: WOLFCOSE_ALG_ML_DSA_44, WOLFCOSE_ALG_ML_DSA_65, or WOLFCOSE_ALG_ML_DSA_87 | | mLDsaKey | Pointer to initialized wolfCrypt ML-DSA key (caller-owned) | | seed | 32-byte ML-DSA

seed (caller-owned), or NULL for public/sign/verify use | | seedLen | Seed length; must be WOLF-COSE_MLDSA_SEED_SZ (32) when seed is non-NULL |

Returns: WOLFCOSE_SUCCESS or error code

Requires: WOLFSSL_HAVE_MLDSA

5.3.8 wc_CoseKey_SetRsa

```
int wc_CoseKey_SetRsa(WOLFCOSE_KEY* key, RsaKey* rsaKey);
```

Associate an RSA key with a COSE key structure.

Parameters: | Name | Description | |——|———| | key | Pointer to initialized COSE key | | rsaKey | Pointer to initialized wolfCrypt RSA key (caller-owned) |

Returns: WOLFCOSE_SUCCESS or error code

Requires: WC_RSA_PSS

5.3.9 wc_CoseKey_SetLms

```
int wc_CoseKey_SetLms(WOLFCOSE_KEY* key, LmsKey* lmsKey);
```

Attach an HSS/LMS stateful hash-based key (RFC 8778) to a COSE key structure. The caller owns the LmsKey; for signing it configures the parameter set and the wolfCrypt private-key read/write callbacks, then calls wc_LmsKey_MakeKey() or wc_LmsKey_Reload() before attaching. Each successful signing call consumes one one-time signature and persists the advanced state through those callbacks. If a signing call fails, wolfCOSE marks the LmsKey bad and it cannot sign again through any attachment until the caller reloads it with wc_LmsKey_Reload().

Parameters: | Name | Description | |——|———| | key | Pointer to initialized COSE key | | lmsKey | Pointer to initialized wolfCrypt LmsKey (caller-owned) |

Returns: WOLFCOSE_SUCCESS or error code

Requires: WOLFSSL_HAVE_LMS (wolfSSL 5.9.2 or later)

5.3.10 wc_CoseKey_SetSymmetric

```
int wc_CoseKey_SetSymmetric(WOLFCOSE_KEY* key, const uint8_t* keyData, size_t
↪ keyLen);
```

Set symmetric key material in a COSE key structure.

Parameters: | Name | Description | |——|———| | key | Pointer to initialized COSE key | | keyData | Pointer to symmetric key bytes (caller-owned buffer) | | keyLen | Length of key in bytes (16, 24, or 32 for AES-GCM) |

Returns: WOLFCOSE_SUCCESS or error code

5.3.11 wc_CoseKey_Encode

```
int wc_CoseKey_Encode(WOLFCOSE_KEY* key, uint8_t* buf, size_t bufSz, size_t*
↳ outLen);
```

Encode a COSE key to CBOR format.

Warning: this serialises the private key when the key has one. `wc_CoseKey_SetEcc()` sets `key->hasPrivate` whenever the attached `ecc_key` is a keypair, and `wc_CoseKey_Encode()` then emits the `-4: d` entry. The output is a perfectly valid COSE_Key, only longer - for P-256 with `alg` set, 112 bytes / map(6) instead of 77 bytes / map(5) - so nothing fails loudly. The same applies to RSA (`-3: d` plus the CRT factors), Ed25519 and Ed448 (`-4: d`), an RFC 9964 AKP key (`-2: priv seed`), and symmetric keys (`-1: k` is the whole key). Anything that publishes a public key derived from a live keypair - WebAuthn/CTAP2 attestation `authData`, ECDH key agreement, JWK-style publication - must use `wc_CoseKey_Encode_ex()` with `WOLFCOSE_KEY_PUBLIC_ONLY`.

Parameters: | Name | Description | |——|———| | `key` | Pointer to COSE key to encode | | `buf` | Output buffer | | `bufSz` | Size of output buffer | | `outLen` | Receives actual encoded length |

Returns: `WOLFCOSE_SUCCESS` or error code

5.3.12 wc_CoseKey_Encode_ex

```
int wc_CoseKey_Encode_ex(const WOLFCOSE_KEY* key, uint8_t* buf, size_t bufSz,
size_t* outLen, uint32_t flags);
```

As `wc_CoseKey_Encode()`, plus output options. Passing `flags = 0` is equivalent to calling `wc_CoseKey_Encode()`.

Flag	Effect
<code>WOLFCOSE_KEY_PUBLIC_ONLY</code>	Emit the public half only. No <code>-4: d</code> for EC2/OKP, no <code>-3: d</code> or CRT factors for RSA, no <code>-2: priv seed</code> for an AKP key.

Unknown flag bits are rejected with `WOLFCOSE_E_INVALID_ARG`.

A symmetric key has no public half - `-1: k` is the key - so `WOLFCOSE_KEY_PUBLIC_ONLY` on `WOLFCOSE_KTY_SYMMETRIC` returns `WOLFCOSE_E_COSE_KEY_TYPE` rather than emitting a key with no key material.

The key structure is not modified; the flag affects only this call, so the same key can still sign afterwards.

```
/* Publish the attestation public key without the private scalar. */
ret = wc_CoseKey_Encode_ex(&key, cose, sizeof(cose), &coseLen,
WOLFCOSE_KEY_PUBLIC_ONLY);
```

Returns: `WOLFCOSE_SUCCESS` or error code

5.3.13 wc_CoseKey_EncodeEccRaw

```
int wc_CoseKey_EncodeEccRaw(int32_t crv,
const uint8_t* x, const uint8_t* y,
const uint8_t* d, size_t coordLen,
```

```
const uint8_t* kid, size_t kidLen, int32_t alg,
uint8_t* out, size_t outSz, size_t* outLen);
```

Encode an EC2 COSE_Key straight from raw affine coordinates. For callers that hold only the coordinates - re-emitting a stored credential record, or echoing a peer key - this avoids `wc_ecc_import_unsigned()`, which pays a full point import, an on-curve check, and an ecc_key worth of stack purely to serialise bytes that are already in hand.

The output is byte-identical to what `wc_CoseKey_Encode_ex()` produces for the same `crv/kid/alg` and the same coordinates.

Parameters: | Name | Description | |——|———| | `crv` | WOLFCOSE_CRV_P256 / P384 / P521 (EC2 curves only) | | `x, y` | Coordinates, each exactly `coordLen` bytes, big-endian, zero-padded | | `d` | Private scalar (`coordLen` bytes), or NULL for a public-only key | | `coordLen` | Must equal the curve size: 32, 48, or 66 | | `kid, kidLen` | Optional key identifier (NULL, 0 to omit) | | `alg` | Algorithm for the 3: `alg` entry, or WOLFCOSE_ALG_UNSET to omit | | `out, outSz, outLen` | Output buffer and encoded length |

Returns: WOLFCOSE_SUCCESS or error code

Nothing here checks that (x, y) is on the curve: the bytes are copied into the map as supplied. Encoding an unvalidated point is safe; *using* one is not, so import it through `wolfCrypt` before any ECDH or verify operation.

There is deliberately no `wc_CoseKey_SetEccRaw()`. WOLFCOSE_KEY holds a single pointer-sized union member for the key, which cannot carry three independent buffers (`x, y, d`), and widening the structure would change `sizeof(WOLFCOSE_KEY)` and break the ABI for already-compiled callers. The free function above is the supported raw-coordinate path.

5.3.14 wc_CoseKey_EncodeSize / wc_CoseKey_EncodeSize_ex

```
int wc_CoseKey_EncodeSize(const WOLFCOSE_KEY* key, size_t* outLen);
int wc_CoseKey_EncodeSize_ex(const WOLFCOSE_KEY* key, size_t* outLen,
uint32_t flags);
```

Compute the **exact** number of bytes `wc_CoseKey_Encode()` / `wc_CoseKey_Encode_ex()` would write, without encoding into a buffer. This is the COSE_Key counterpart of `wc_CoseSign1_SignSize_ex()`: nothing is written and no key material is exported, only the component lengths of the attached key are read. The result is exact, not an upper bound, so it can size a buffer or reject an oversized key before committing storage.

`flags` takes the same WOLFCOSE_KEY_* values as `wc_CoseKey_Encode_ex()`, so WOLFCOSE_KEY_PUBLIC_ONLY sizes the public-only encoding. `flags = 0` sizes the private encoding - see the warning on `wc_CoseKey_Encode()`.

Returns: WOLFCOSE_SUCCESS or error code

5.3.15 wc_CoseKey_Decode

```
int wc_CoseKey_Decode(WOLFCOSE_KEY* key, const uint8_t* buf, size_t bufSz);
```

Decode a COSE key from CBOR format.

Parameters: | Name | Description | |——|———| | `key` | Initialized COSE key structure; may have a `wolfCrypt` key attached | | `buf` | Input CBOR buffer | | `bufSz` | Size of input buffer |

Returns: WOLFCOSE_SUCCESS or error code

Attach a wolfCrypt key with `wc_CoseKey_SetEcc()`, `wc_CoseKey_SetEd25519()`, `wc_CoseKey_SetEd448()`, `wc_CoseKey_SetRsa()`, `wc_CoseKey_SetMlDsa()`, or `wc_CoseKey_SetSymmetric()`. These record which wolfCrypt object is attached. Without an attachment, supported asymmetric metadata is still validated and returned, but key material is not imported. Assigning the key.* union directly records no attached type and imports nothing. Only attachments recorded by a setter are preserved across decode; an untyped key.* union is cleared before parsing.

When a key is attached, the decoded `kty/crv` must name that key type or `WOLFCOSE_E_COSE_KEY_TYPE` is returned before any importer runs. To learn which key type a buffer holds before attaching anything, use `wc_CoseKey_PeekInfo()`.

Decoding is strict: preferred CBOR only, integer labels only, no duplicate labels, and `bufSz` must be exactly the encoded length. COSE_Key text labels are rejected even though COSE header maps permit text labels. See [Getting Started - Strict decoding](#). Keys containing the optional `key_ops` label (4) return `WOLFCOSE_E_UNSUPPORTED` before any key material is imported because the fixed-size WOLF-COSE_KEY wrapper cannot retain arbitrary operation arrays.

An attached `ecc_key` that will receive private EC2 material must be freshly initialized, with no existing curve or key material. Reusing a populated ECC object returns `WOLFCOSE_E_INVALID_ARG` without replacing its key. Free and initialize the object again before decoding another private EC2 key. This precondition lets wolfCOSE roll back a failed software import without losing caller-owned wolfCrypt configuration. Backends or math layouts that cannot be rolled back safely return `WOLFCOSE_E_UNSUPPORTED` before private import.

5.3.16 wc_CoseKey_PeekInfo

```
typedef struct WOLFCOSE_KEY_INFO {
    int32_t      kty;      /* WOLFCOSE_KTY_*, always set on success */
    int32_t      alg;      /* WOLFCOSE_ALG_*, WOLFCOSE_ALG_UNSET if absent */
    int32_t      crv;      /* WOLFCOSE_CRV_*, 0 if absent or N/A */
    const uint8_t* kid;    /* Key ID, zero-copy pointer, NULL if absent */
    size_t      kidLen;
} WOLFCOSE_KEY_INFO;
```

```
int wc_CoseKey_PeekInfo(const uint8_t* in, size_t inSz,
                       WOLFCOSE_KEY_INFO* info);
```

Read `kty`, `alg`, `crv`, and `kid` out of a COSE_Key buffer without importing any key material and without needing a wolfCrypt key object.

When asymmetric key material is to be imported, peek first to identify its type, then attach the matching wolfCrypt object once:

```
WOLFCOSE_KEY_INFO info;
```

```
ret = wc_CoseKey_PeekInfo(in, inSz, &info);
if (ret == WOLFCOSE_SUCCESS) {
    if (info.kty == WOLFCOSE_KTY_EC2) {
        (void)wc_ecc_init(&ecc);
        ret = wc_CoseKey_SetEcc(&key, info.crv, &ecc);
    }
    else if (info.kty == WOLFCOSE_KTY_OKP) {
        (void)wc_ed25519_init(&ed);
        ret = wc_CoseKey_SetEd25519(&key, &ed);
    }
}
```

```

/* ... */
if (ret == WOLFCOSE_SUCCESS) {
    ret = wc_CoseKey_Decode(&key, in, inSz);
}
}

```

in is not modified and nothing is consumed, so the call is repeatable. kid points into in, so it stays valid only as long as that buffer does.

The same structural checks wc_CoseKey_Decode() applies are applied here: integer labels only, no duplicates, kty required, and no trailing bytes. Label -1 is crv for EC2/OKP but k/n for symmetric/RSA keys; the value is dispatched on its CBOR type, so crv stays 0 for the latter. A key_ops label returns WOLFCOSE_E_UNSUPPORTED, matching decode. On any error every field of info is cleared.

Returns: WOLFCOSE_SUCCESS or error code

5.4 COSE_Sign1 API

5.4.1 wc_CoseSign1_Sign

```

int wc_CoseSign1_Sign(
    const WOLFCOSE_KEY* key,
    int32_t alg,
    const uint8_t* kid, size_t kidLen,
    const uint8_t* payload, size_t payloadLen,
    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen,
    WC_RNG* rng
);

```

Create a COSE_Sign1 message (single signer).

Parameters: | Name | Description | |——|———| | key | Signing key with private material, or an external signing callback when enabled | | alg | Algorithm: WOLFCOSE_ALG_ESP256, WOLFCOSE_ALG_ESP384, WOLFCOSE_ALG_ESP512, WOLFCOSE_ALG_ED25519, WOLFCOSE_ALG_ED448, etc. The RFC 9053 WOLFCOSE_ALG_ES256/ES384/ES512/EDDSA IDs need WOLFCOSE_ENABLE_DEPRECATED_ALGS. | | kid, kidLen | Optional key identifier | | payload, payloadLen | Payload to include in message (or NULL for detached) | | detachedPayload, detachedPayloadLen | Payload to sign but not include | | extAad, extAadLen | External additional authenticated data | | scratch, scratchSz | Scratch buffer (min WOLFCOSE_MAX_SCRATCH_SZ) | | out, outSz, outLen | Output buffer and length | | rng | Random number generator for local signing; may be NULL with an external signing callback |

Returns: WOLFCOSE_SUCCESS or error code

5.4.2 wc_CoseSign1_Sign_ex

```

int wc_CoseSign1_Sign_ex(
    WOLFCOSE_KEY* key,
    int32_t alg,
    const uint8_t* kid, size_t kidLen,
    const uint8_t* payload, size_t payloadLen,

```

```

    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen,
    WC_RNG* rng, uint32_t flags
);

```

As `wc_CoseSign1_Sign()`, plus output options. Passing `flags = 0` is equivalent to calling `wc_CoseSign1_Sign()`.

Flag	Effect
<code>WOLFCOSE_SIGN1_UNTAGGED</code>	Omit the CBOR tag 18 prefix, so output starts with the four-element array (0x84).

The tag is not covered by the signature. A caller emitting untagged output must establish the message type out of band.

Returns: `WOLFCOSE_SUCCESS` or error code

5.4.3 wc_CoseSign1_SignSize_ex

```

int wc_CoseSign1_SignSize_ex(
    const WOLFCOSE_KEY* key,
    int32_t alg,
    size_t kidLen,
    size_t payloadLen,
    size_t detachedLen,
    uint32_t flags,
    size_t* outLen
);

```

Computes the exact encoded size that `wc_CoseSign1_Sign_ex()` would produce without signing. It does not use an RNG, private key operation, or external signer callback. The data itself is not required because only `kidLen`, `payloadLen`, and `detachedLen` affect the framing.

`key` may be `NULL` when the algorithm fixes the signature length. It is required for RSA-PSS, for HSS-LMS, and for the deprecated `WOLFCOSE_ALG_EDDSA` when both `Ed25519` and `Ed448` are enabled. `WOLFCOSE_ALG_ED25519` and `WOLFCOSE_ALG_ED448` each pin a length, so they never need a key.

Returns: `WOLFCOSE_SUCCESS` or error code

5.4.4 wc_CoseSign1_Verify

```

int wc_CoseSign1_Verify(
    const WOLFCOSE_KEY* key,
    const uint8_t* coseMsg, size_t coseMsgLen,
    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* hdr,
    const uint8_t** payload, size_t* payloadLen
);

```

Verify a COSE_Sign1 message and extract payload.

Parameters: | Name | Description | |——|———| | key | Verification key (public key sufficient) | | coseMsg, coseMsgLen | COSE_Sign1 message to verify | | detachedPayload, detachedPayloadLen | Detached payload (if message has null payload) | | extAad, extAadLen | External AAD (must match what was used during signing) | | scratch, scratchSz | Scratch buffer | | hdr | Receives parsed header information | | payload, payloadLen | Receives pointer to payload (zero-copy into coseMsg) |

Returns: WOLFCOSE_SUCCESS, WOLFCOSE_E_COSE_SIG_FAIL, or other error

5.5 COSE Countersignature API

These RFC 9338 APIs operate on tagged COSE_Sign1, COSE_Sign, COSE_Encrypt0, COSE_Encrypt, COSE_Mac0, and COSE_Mac messages. A countersignature authenticates the target message's protected header, payload, and existing cryptographic output where applicable.

Countersignatures support ECDSA, EdDSA, RSA-PSS, and ML-DSA. WOLFCOSE_ALG_HSS_LMS is rejected with WOLFCOSE_E_COSE_BAD_ALG before any signer runs, and a build whose only signature algorithm is HSS/LMS does not enable the feature.

5.5.1 wc_Cose_AddCounterSignature

```
int wc_Cose_AddCounterSignature(
    const WOLFCOSE_COUNTERSIGNATURE* counterSigner,
    const uint8_t* in, size_t inSz,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen,
    WC_RNG* rng
);
```

Add a full V2 countersignature in unprotected header parameter 11. Existing full countersignatures are retained and the new value is appended. out may equal in for exact in-place growth; other overlapping buffers are rejected. scratch must be disjoint from all input and output buffers.

5.5.2 wc_Cose_AddCounterSignature0

```
int wc_Cose_AddCounterSignature0(
    const WOLFCOSE_COUNTERSIGNATURE0* counterSigner,
    const uint8_t* in, size_t inSz,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen,
    WC_RNG* rng
);
```

Add one abbreviated V2 countersignature in unprotected header parameter 12. Only one abbreviated countersignature is allowed per target. scratch must be disjoint from all input and output buffers.

5.5.3 wc_Cose_VerifyCounterSignature

```
int wc_Cose_VerifyCounterSignature(
    const WOLFCOSE_KEY* key, size_t counterIndex,
    const uint8_t* in, size_t inSz,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* counterHdr
);
```

Verify a full countersignature selected by zero-based index and return its parsed headers. V2 label 11 and legacy RFC 8152 label 7 are accepted. V2 is preferred when both are present. If the countersignature algorithm is carried only in the unprotected header bucket, key->alg must pin the same algorithm; an unset or mismatched key policy is rejected. scratch must be disjoint from the message, detached payload, external AAD, and counterHdr.

5.5.4 wc_Cose_VerifyCounterSignature0

```
int wc_Cose_VerifyCounterSignature0(
    const WOLFCOSE_COUNTERSIGNATURE0* counterSigner,
    const uint8_t* in, size_t inSz,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz
);
```

Verify an abbreviated countersignature using the algorithm and key supplied by the application. V2 label 12 and legacy RFC 8152 label 9 are accepted. scratch must be disjoint from the message, detached payload, and external AAD.

All four APIs require the detached payload or ciphertext, when used by the target, to match the original operation. Verification must also receive the same countersignature external AAD that was supplied when the countersignature was created; it is independent of any AAD the target operation used. They return WOLFCOSE_SUCCESS or a negative wolfCOSE error code.

5.6 COSE_Encrypt0 API

5.6.1 wc_CoseEncrypt0_Encrypt

```
int wc_CoseEncrypt0_Encrypt(
    const WOLFCOSE_KEY* key,
    int32_t alg,
    const uint8_t* iv, size_t ivLen,
    const uint8_t* payload, size_t payloadLen,
    uint8_t* detachedCt, size_t detachedCtSz, size_t* detachedCtLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen
);
```

Create a COSE_Encrypt0 message.

Parameters: | Name | Description | |——|———| | key | Symmetric encryption key | | alg | Algorithm: WOLFCOSE_ALG_A128GCM, WOLFCOSE_ALG_A192GCM, WOLFCOSE_ALG_A256GCM, etc. |

| iv, ivLen | Initialization vector (12 bytes for AES-GCM) | | payload, payloadLen | Plaintext to encrypt | | detachedCt, detachedCtSz, detachedCtLen | Optional: receive ciphertext separately | | extAad, extAadLen | External additional authenticated data | | scratch, scratchSz | Scratch buffer | | out, outSz, outLen | Output buffer and length |

Returns: WOLFCOSE_SUCCESS or error code

5.6.2 wc_CoseEncrypt0_Decrypt

```
int wc_CoseEncrypt0_Decrypt(
    const WOLFCOSE_KEY* key,
    const uint8_t* coseMsg, size_t coseMsgLen,
    const uint8_t* detachedCt, size_t detachedCtLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* hdr,
    uint8_t* plaintext, size_t plaintextSz, size_t* plaintextLen
);
```

Decrypt a COSE_Encrypt0 message.

Parameters: | Name | Description | |——|———| | key | Symmetric decryption key | | coseMsg, coseMsgLen | COSE_Encrypt0 message | | detachedCt, detachedCtLen | Detached ciphertext (if message has null ciphertext) | | extAad, extAadLen | External AAD (must match encryption) | | scratch, scratchSz | Scratch buffer | | hdr | Receives parsed header information | | plaintext, plaintextSz, plaintextLen | Output buffer for decrypted data |

Returns: WOLFCOSE_SUCCESS, WOLFCOSE_E_COSE_DECRYPT_FAIL, or other error

5.7 COSE_Mac0 API

5.7.1 wc_CoseMac0_Create

```
int wc_CoseMac0_Create(
    WOLFCOSE_KEY* key,
    int32_t alg,
    const uint8_t* kid, size_t kidLen,
    const uint8_t* payload, size_t payloadLen,
    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen
);
```

Create a COSE_Mac0 message.

Parameters: | Name | Description | |——|———| | key | Symmetric MAC key | | alg | Algorithm: WOLFCOSE_ALG_HMAC_256_256, WOLFCOSE_ALG_AES_MAC_128_64, etc. | | kid, kidLen | Key identifier (can be NULL, 0) | | payload, payloadLen | Payload to include in message | | detachedPayload, detachedPayloadLen | Payload to MAC but not include | | extAad, extAadLen | External AAD | | scratch, scratchSz | Scratch buffer | | out, outSz, outLen | Output buffer and length |

Returns: WOLFCOSE_SUCCESS or error code

5.7.2 wc_CoseMac0_Verify

```
int wc_CoseMac0_Verify(
    const WOLFCOSE_KEY* key,
    const uint8_t* coseMsg, size_t coseMsgLen,
    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* hdr,
    const uint8_t** payload, size_t* payloadLen
);
```

Verify a COSE_Mac0 message.

Parameters: | Name | Description | |---|-----| | key | Symmetric MAC key | | coseMsg, coseMsgLen | COSE_Mac0 message | | detachedPayload, detachedPayloadLen | Detached payload if applicable | | extAad, extAadLen | External AAD | | scratch, scratchSz | Scratch buffer | | hdr | Receives parsed header | | payload, payloadLen | Receives payload pointer |

Returns: WOLFCOSE_SUCCESS, WOLFCOSE_E_MAC_FAIL, or other error

5.8 COSE_Sign API (Multi-Signer)

5.8.1 wc_CoseSign_Sign

```
int wc_CoseSign_Sign(
    const WOLFCOSE_SIGNATURE* signers, size_t signerCount,
    const uint8_t* payload, size_t payloadLen,
    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen,
    WC_RNG* rng
);
```

Create a COSE_Sign message with multiple signers.

Parameters: | Name | Description | |---|-----| | signers | Array of WOLFCOSE_SIGNATURE structures | | signerCount | Number of signers | | Other parameters | Same as wc_CoseSign1_Sign |

5.8.2 wc_CoseSign_Verify

```
int wc_CoseSign_Verify(
    const WOLFCOSE_KEY* key,
    size_t signerIdx,
    const uint8_t* coseMsg, size_t coseMsgLen,
    const uint8_t* detachedPayload, size_t detachedPayloadLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* hdr,
    const uint8_t** payload, size_t* payloadLen
);
```

Verify one signer's signature in a COSE_Sign message.

Parameters: | Name | Description | |——|———| | key | Verification key for the specific signer | | signerIdx | Zero-based index of signer to verify | | Other parameters | Same as wc_CoseSign1_Verify |

5.9 COSE_Encrypt API (Multi-Recipient)

5.9.1 wc_CoseEncrypt_Encrypt

```
int wc_CoseEncrypt_Encrypt(
    const WOLFCOSE_RECIPIENT* recipients, size_t recipientCount,
    int32_t contentAlgId,
    const uint8_t* iv, size_t ivLen,
    const uint8_t* payload, size_t payloadLen,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen,
    WC_RNG* rng
);
```

Create a COSE_Encrypt message for multiple recipients.

Direct recipients are encoded with the mandatory unprotected {1: -6} algorithm header and an empty bstr ciphertext item. This adds two bytes per direct recipient compared with the former empty-map encoding; callers using fixed output buffers must leave room for it.

Parameters: | Name | Description | |——|———| | recipients | Array of WOLFCOSE_RECIPIENT structures with keys and algorithms | | recipientCount | Number of recipients | | contentAlgId | Content encryption algorithm (e.g., WOLFCOSE_ALG_A128GCM) | | iv, ivLen | Initialization vector (12 bytes for AES-GCM) | | payload, payloadLen | Plaintext to encrypt (inline) | | detachedPayload, detachedLen | Plaintext to encrypt but not include in message | | extAad, extAadLen | External AAD | | scratch, scratchSz | Scratch buffer | | out, outSz, outLen | Output buffer | | rng | Random number generator |

Returns: WOLFCOSE_SUCCESS or error code

5.9.2 wc_CoseEncrypt_Decrypt

```
int wc_CoseEncrypt_Decrypt(
    const WOLFCOSE_RECIPIENT* recipient,
    size_t recipientIndex,
    const uint8_t* in, size_t inSz,
    const uint8_t* detachedCt, size_t detachedCtLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* hdr,
    uint8_t* plaintext, size_t plaintextSz, size_t* plaintextLen
);
```

Decrypt a COSE_Encrypt message as a specific recipient.

Every recipient must declare its key-management algorithm. Direct encryption algorithms may coexist with each other, direct key agreement requires one recipient, and key-transport algorithms may

coexist with each other. Direct recipients must carry an empty bstr or null ciphertext item. If the body algorithm is only in the unprotected header, recipient->key->alg must pin the same algorithm.

Parameters: | Name | Description | |——|———| | recipient | Recipient structure with key and algorithm | | recipientIndex | Zero-based index of recipient in message | | in, inSz | COSE_Encrypt message | | detachedCt, detachedCtLen | Detached ciphertext (if applicable) | | extAad, extAadLen | External AAD (must match encryption) | | scratch, scratchSz | Scratch buffer | | hdr | Receives parsed header | | plaintext, plaintextSz, plaintextLen | Output buffer |

Returns: WOLFCOSE_SUCCESS, WOLFCOSE_E_COSE_DECRYPT_FAIL, or other error

5.10 COSE_Mac API (Multi-Recipient)

5.10.1 wc_CoseMac_Create

```
int wc_CoseMac_Create(
    const WOLFCOSE_RECIPIENT* recipients, size_t recipientCount,
    int32_t macAlgId,
    const uint8_t* payload, size_t payloadLen,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    uint8_t* out, size_t outSz, size_t* outLen
);
```

Create a COSE_Mac message for multiple recipients.

Each direct recipient is encoded with the mandatory unprotected {1: -6} algorithm header and an empty bstr ciphertext item. This adds two bytes per recipient compared with the former empty-map encoding.

Parameters: | Name | Description | |——|———| | recipients | Array of WOLFCOSE_RECIPIENT structures | | recipientCount | Number of recipients | | macAlgId | MAC algorithm (e.g., WOLFCOSE_ALG_HMAC_256_256) | | payload, payloadLen | Payload to include in message | | detachedPayload, detachedLen | Payload to MAC but not include | | extAad, extAadLen | External AAD | | scratch, scratchSz | Scratch buffer | | out, outSz, outLen | Output buffer |

Returns: WOLFCOSE_SUCCESS or error code

5.10.2 wc_CoseMac_Verify

```
int wc_CoseMac_Verify(
    const WOLFCOSE_RECIPIENT* recipient,
    size_t recipientIndex,
    const uint8_t* in, size_t inSz,
    const uint8_t* detachedPayload, size_t detachedLen,
    const uint8_t* extAad, size_t extAadLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_HDR* hdr,
    const uint8_t** payload, size_t* payloadLen
);
```

Verify a COSE_Mac message as a specific recipient.

Every recipient must declare WOLFCOSE_ALG_DIRECT, sibling recipients must use direct mode, and the recipient ciphertext item must be an empty bstr or null. If the body algorithm is only in the unprotected header, recipient->key->alg must pin the same algorithm.

Parameters: | Name | Description | |——|———| | recipient | Recipient structure with key and algorithm | | recipientIndex | Zero-based index of recipient in message | | in, inSz | COSE_Mac message | | detachedPayload, detachedLen | Detached payload (if applicable) | | extAad, extAadLen | External AAD | | scratch, scratchSz | Scratch buffer | | hdr | Receives parsed header | | payload, payloadLen | Receives payload pointer |

Returns: WOLFCOSE_SUCCESS, WOLFCOSE_E_MAC_FAIL, or other error

5.11 PSA/EAT API

Available only when WOLFCOSE_ENABLE_EAT_PSA and the selected profile and envelope gates are defined. Include <wolfcase/eat_psa.h>. See [PSA-EAT](#) for configuration and security requirements.

5.11.1 wc_CoseEatPsaToken_Verify

```
int wc_CoseEatPsaToken_Verify(const WOLFCOSE_KEY* key,
    const uint8_t* in, size_t inSz,
    const uint8_t* expectedNonce, size_t expectedNonceLen,
    uint8_t* scratch, size_t scratchSz,
    WOLFCOSE_EAT_PSA_TOKEN* token);
```

Authenticates a tagged, attached RFC 9783 current or selected legacy token, checks profile-required claim structure and the expected nonce, then returns zero-copy claim spans in token. The input buffer must remain unchanged while the output token is used. expectedNonce is required and must be 32, 48, or 64 bytes.

5.11.2 Other PSA/EAT entry points

Function	Required feature gate	Purpose
wc_CoseEatPsaToken_WOLF_COSE_ENABLE_EAT_PSA_ISSUE	WOLFCOSE_ENABLE_EAT_PSA_ISSUE	Encode current-profile claims
wc_CoseEatPsaToken_WOLF_COSE_ENABLE_EAT_PSA_SIGN1_ISSUE	WOLFCOSE_ENABLE_EAT_PSA_SIGN1_ISSUE plus common issue	Encode and create a current Sign1 token
wc_CoseEatPsaToken_WOLF_COSE_ENABLE_EAT_PSA_MAC0_ISSUE	WOLFCOSE_ENABLE_EAT_PSA_MAC0_ISSUE plus common issue	Encode and create a current Mac0 token
wc_CoseEatPsaToken_WOLF_COSE_ENABLE_EAT_PSA_UAID_RESOLVER	WOLFCOSE_ENABLE_EAT_PSA_UAID_RESOLVER	Resolve a candidate key from an untrusted UAID, then authenticate the original token
wc_CoseEatPsaToken_WOLF_COSE_ENABLE_EAT_PSA_COMPONENT_ITERATOR	WOLFCOSE_ENABLE_EAT_PSA_COMPONENT_ITERATOR	Decode authenticated software components one at a time

The three writable buffers passed to either creation API (`claimsBuf`, `scratch`, and `out`) must be pairwise disjoint. Exact or partial overlap is rejected with `WOLFCOSE_E_INVALID_ARG` before claims are encoded. `claimsBuf` must additionally be disjoint from the claims structure, component array, and every nonempty input span. The direct `EncodeClaims` output has the same input-disjointness requirement; in-place encoding is rejected.

The raw-key verifier rejects `x5chain`; validate certificates outside this API before supplying a public key. Verification provides claims for caller policy appraisal and does not itself authorize a device.

`wc_CoseEatPsaToken_ForEachComponent()` supplies a component structure that is valid only for its callback. The structure's span data borrows from the verified input token and remains valid while that input remains unchanged.

5.12 CBOR API

Basic CBOR encoding/decoding functions in `wolfcose.h`:

5.12.1 Context Setup

`WOLFCOSE_CBOR_CTX` carries a mutable `buf` for encode output and a `const cbuf` for decode input. The initializers set the appropriate pointer, clear the opposite pointer, set `bufSz`, and zero `idx` in a single call:

```
int wc_CBOR_EncoderInit(WOLFCOSE_CBOR_CTX* ctx, uint8_t* buf, size_t bufSz);
int wc_CBOR_DecomderInit(WOLFCOSE_CBOR_CTX* ctx, const uint8_t* buf, size_t
    ↪ bufSz);
```

Use these initializers instead of assigning context fields directly. Public CBOR decoding always requires RFC 8949 preferred serialization. The optional PSA/EAT verifier handles RFC 9783's permitted definite-length variation serialization privately; indefinite-length forms remain unsupported.

```
WOLFCOSE_CBOR_CTX ctx;
```

```
(void)wc_CBOR_EncoderInit(&ctx, out, sizeof(out));
ret = wc_CBOR_EncodeMapStart(&ctx, 2);
...
(void)wc_CBOR_DecomderInit(&ctx, in, inSz);
ret = wc_CBOR_DecodeMapStart(&ctx, &count);
```

Returns: `WOLFCOSE_SUCCESS`, or `WOLFCOSE_E_INVALID_ARG` if `ctx` or `buf` is `NULL`.

5.12.2 Encoding Functions

Function	Description
<code>wc_CBOR_EncoderInit(ctx, buf, bufSz)</code>	Initialize an encode context
<code>wc_CBOR_EncodeUint(ctx, val)</code>	Encode unsigned integer
<code>wc_CBOR_EncodeInt(ctx, val)</code>	Encode signed integer
<code>wc_CBOR_EncodeBstr(ctx, data, len)</code>	Encode byte string
<code>wc_CBOR_EncodeTstr(ctx, str, len)</code>	Encode text string
<code>wc_CBOR_EncodeArrayStart(ctx, count)</code>	Encode array header
<code>wc_CBOR_EncodeMapStart(ctx, count)</code>	Encode map header
<code>wc_CBOR_EncodeTag(ctx, tag)</code>	Encode CBOR tag
<code>wc_CBOR_EncodeNull(ctx)</code>	Encode null

Function	Description
<code>wc_CBOR_EncodeTrue(ctx) / wc_CBOR_EncodeFalse(ctx)</code>	Encode booleans

5.12.3 Decoding Functions

Function	Description
<code>wc_CBOR_DecoderInit(ctx, buf, bufSz)</code>	Initialize a decode context
<code>wc_CBOR_DecodeUInt(ctx, val)</code>	Decode unsigned integer
<code>wc_CBOR_DecodeInt(ctx, val)</code>	Decode signed integer
<code>wc_CBOR_DecodeBstr(ctx, data, len)</code>	Decode byte string (zero-copy)
<code>wc_CBOR_DecodeTstr(ctx, str, len)</code>	Decode text string (zero-copy)
<code>wc_CBOR_DecodeArrayStart(ctx, count)</code>	Decode array header
<code>wc_CBOR_DecodeMapStart(ctx, count)</code>	Decode map header
<code>wc_CBOR_DecodeTag(ctx, tag)</code>	Decode CBOR tag
<code>wc_CBOR_DecodeLabel(ctx, label)</code>	Decode an int-or-text map label
<code>wc_CBOR_Skip(ctx)</code>	Skip over any CBOR item
<code>wc_CBOR_SkipItem(ctx, data, dataLen)</code>	Skip an item and capture its raw bytes
<code>wc_CBOR_PeekType(ctx)</code>	Peek at next item's major type

Decoding is strict by default. Ordinary decode entry points require RFC 8949 Section 4.2.1 preferred (shortest-form) arguments and reject indefinite lengths. The optional PSA/EAT verifier privately admits the non-preferred definite-length forms RFC 9783 requires; it never admits indefinite lengths. `wc_CBOR_DecodeHead()`, `wc_CBOR_DecodeLabel()`, `wc_CBOR_Skip()`, and `wc_CBOR_SkipItem()` validate every text string they encounter, even when merely traversing it, and return `WOLFCOSE_E_CBOR_MALFORMED` for invalid UTF-8. See [Getting Started - Strict decoding](#) before debugging an interop failure.

5.12.4 wc_CBOR_SkipItem

```
int wc_CBOR_SkipItem(WOLFCOSE_CBOR_CTX* ctx,
                    const uint8_t** data, size_t* dataLen);
```

As `wc_CBOR_Skip()`, but also reports the raw encoded bytes of the item that was skipped, zero-copy into the decoder input. This is the deferred/nested parse primitive: capture a sub-item now, parse it later or with a different decoder. It replaces the hand-rolled

```
start = ctx.idx;
ret = wc_CBOR_Skip(&ctx);
ptr = ctx.cbuf + start;
len = ctx.idx - start;
```

that every integrator ends up writing - CTAP2 allowList entries, a COSE_Key embedded in an extension map, keyAgreement in authenticatorClientPIN.

```
const uint8_t* sub;
size_t subLen;
```

```
ret = wc_CBOR_SkipItem(&ctx, &sub, &subLen);
if (ret == WOLFCOSE_SUCCESS) {
    ret = wc_CoseKey_Decode(&peerKey, sub, subLen); /* parse it later */
}
```

wc_CBOR_Skip() and wc_CBOR_SkipItem() validate UTF-8 in every text string they traverse. On failure the capture outputs are untouched.

Returns: WOLFCOSE_SUCCESS or error code

5.12.5 wc_CBOR_DecodeLabel

```
typedef struct WOLFCOSE_CBOR_LABEL {
    int64_t    val;          /* Integer label, valid when isText == 0 */
    const uint8_t* text;     /* Text label, points into the input buffer */
    size_t     textLen;
    uint8_t    isText;
} WOLFCOSE_CBOR_LABEL;
```

```
int wc_CBOR_DecodeLabel(WOLFCOSE_CBOR_CTX* ctx, WOLFCOSE_CBOR_LABEL* label);
int wc_CBOR_LabelIsInt(const WOLFCOSE_CBOR_LABEL* label, int64_t val);
int wc_CBOR_LabelIsText(const WOLFCOSE_CBOR_LABEL* label,
                        const uint8_t* text, size_t textLen);
```

RFC 9052 defines label = int / tstr. Applications may use either form for their own map parameters, while registered COSE parameters retain their specified numeric labels. wc_CBOR_DecodeLabel() consumes one item and reports whichever form it found, so a parser writes the dispatch once instead of duplicating a wc_CBOR_PeekType() branch at every map.

Major type 3 validates UTF-8 and, when valid, fills text/textLen with isText == 1 and no copy. Invalid UTF-8 returns WOLFCOSE_E_CBOR_MALFORMED; any other major type returns WOLFCOSE_E_CBOR_TYPE.

wc_CBOR_LabelIsInt() and wc_CBOR_LabelIsText() return 1 on match and 0 otherwise, including for a NULL label. Text comparison is byte-exact: no Unicode normalization or case folding, matching how CTAP2 and COSE compare labels.

```
WOLFCOSE_CBOR_LABEL label;
static const uint8_t algText[] = "alg";
```

```
ret = wc_CBOR_DecodeLabel(&ctx, &label);
if (wc_CBOR_LabelIsInt(&label, 3) || wc_CBOR_LabelIsText(&label, algText, 3)) {
    ret = wc_CBOR_DecodeInt(&ctx, &alg);
}
else {
    ret = wc_CBOR_Skip(&ctx);
}
```

COSE header parsers accept both forms and bitwise track text-label duplicates, including between protected and unprotected header buckets. Registered COSE parameters remain numeric: a text label such as "alg" is an unknown extension, not an alias for numeric label 1. Unknown non-critical extensions are skipped; an unknown entry listed in numeric crit is rejected. wc_CoseKey_Decode() and wc_CoseKey_PeekInfo() always require integer labels.

Returns: WOLFCOSE_SUCCESS or error code

5.13 Error Codes

Code	Name	Description
0	WOLFCOSE_SUCCESS	Operation completed successfully
-9000	WOLFCOSE_E_INVALID_ARG	Invalid argument (NULL pointer, etc.)
-9001	WOLFCOSE_E_BUFFER_TOO_SMALL	Output buffer insufficient
-9002	WOLFCOSE_E_CBOR_MALFORMED	CBOR parsing error
-9003	WOLFCOSE_E_CBOR_TYPE	Unexpected CBOR type
-9004	WOLFCOSE_E_CBOR_OVERFLOW	Integer overflow in CBOR
-9006	WOLFCOSE_E_CBOR_DEPTH	CBOR nesting too deep
-9010	WOLFCOSE_E_COSE_BAD_TAG	Wrong COSE tag for message type
-9011	WOLFCOSE_E_COSE_BAD_ALG	Unsupported or invalid algorithm
-9012	WOLFCOSE_E_COSE_SIG_FAIL	Signature verification failed
-9013	WOLFCOSE_E_COSE_DECRYPT_FAIL	Decrypt failed/authentication failed
-9014	WOLFCOSE_E_COSE_BAD_HEADER	Invalid COSE header
-9015	WOLFCOSE_E_COSE_KEY_WRONG	Wrong key type for operation
-9016	WOLFCOSE_E_COSE_MAC_FAIL	COSE MAC verification failed
-9020	WOLFCOSE_E_CRYPT0	wolfCrypt error
-9021	WOLFCOSE_E_UNSUPPORTED_FEATURE	Feature not supported
-9022	WOLFCOSE_E_MAC_FAIL	MAC verification failed
-9023	WOLFCOSE_E_DETACHED_PAYLOAD	Detached payload required but not provided
-9030	WOLFCOSE_E_EAT_PSA_CLAIM	PSA/EAT required claim is malformed, missing, or duplicated (PSA/EAT builds only)
-9031	WOLFCOSE_E_EAT_PSA_PROFILE	Profile does not match a selected PSA/EAT profile (PSA/EAT builds only)
-9032	WOLFCOSE_E_EAT_PSA_NONCE	Authenticated token nonce does not match the expected nonce (PSA/EAT builds only)
-9033	WOLFCOSE_E_EAT_PSA_KEY	PSA/EAT key resolution failed (PSA/EAT builds only)

5.14 See Also

- [Getting Started](#): Build instructions and examples
- [Algorithms](#): Supported algorithms
- [Macros](#): Compile-time configuration
- [PSA-EAT](#): RFC 9783 PSA attestation support

6 Configuration Macros

wolfCOSE has two configuration modes. The default is an opt-out full build: every algorithm wolfSSL provides is enabled, and you strip what you don't need with `WOLFCOSE_NO_*` defines. Alternatively, `WOLFCOSE_LEAN` switches to an opt-in core build and you add extensions with `WOLFCOSE_ENABLE_*`. See [Lean Configuration Layer](#).

Draft, pre-RFC features are held behind a separate acknowledgement, `WOLFCOSE_EXPERIMENTAL`; see [Experimental Features](#).

6.1 Experimental Features

Some COSE work is standardized in an IETF **Internet-Draft that is not yet a finalized RFC** (for example COSE-HPKE and ML-KEM key encapsulation). While a spec is a draft, its wire format and API may still change, so wolfCOSE keeps that code off in every normal build and requires you to opt in twice — once for the master acknowledgement, once for the specific feature.

Define	Role
<code>WOLFCOSE_EXPERIMENTAL</code>	Master acknowledgement. Enables no functionality on its own; it only <i>permits</i> the individually selected experimental features below.
<code>WOLFCOSE_ENABLE_<X></code>	Opt in one experimental feature (each keeps its own fine-grained gate).

An experimental feature is compiled in only when **both** its own `WOLFCOSE_ENABLE_<X>` and `WOLFCOSE_EXPERIMENTAL` are defined. Selecting the feature without the acknowledgement is a hard compile error:

```
#if defined(WOLFCOSE_ENABLE_<X>) && !defined(WOLFCOSE_EXPERIMENTAL)
    #error "WOLFCOSE_ENABLE_<X> selects experimental draft code (spec not yet a
    ↪ finalized RFC); also define WOLFCOSE_EXPERIMENTAL to acknowledge"
#endif
```

`WOLFCOSE_ENABLE_EXPERIMENTAL_EXAMPLE` is a permanent reference exemplar that enables no functionality; it documents the pattern above and is the target the `make experimental-check` gate exercises. Compile examples:

```
# Normal build: zero experimental code, no acknowledgement needed.
cc ... src/*.c

# Enable an experimental feature (requires both defines):
cc -DWOLFCOSE_EXPERIMENTAL -DWOLFCOSE_ENABLE_EXPERIMENTAL_EXAMPLE ...

# Feature without acknowledgement -> compile error, by design.
cc -DWOLFCOSE_ENABLE_EXPERIMENTAL_EXAMPLE ...
```

Warning: experimental features track drafts still in flux. Their message wire format, header parameters, and public API may change or be removed between wolfCOSE releases with no compatibility guarantee. Do not depend on them in a stable deployment.

Graduation policy. When a draft is published as an RFC, its `WOLFCOSE_EXPERIMENTAL` requirement is removed in a focused follow-up and the feature becomes an ordinary gate (default-on full build, `WOLFCOSE_ENABLE_<X>` under `WOLFCOSE_LEAN`, `WOLFCOSE_NO_<X>` to strip), following the [Algorithm Gates](#) convention.

COSE-HPKE is currently an experimental feature. It stays off in every build and requires the master acknowledgement plus the relevant operation gate. See [Experimental Features](#) for its current draft status, scope, and graduation plan.

6.2 Lean Configuration Layer

Defining WOLFCOSE_LEAN keeps only the core (COSE_Sign1/Encrypt0/Mac0 with ESP256, AES-GCM, and HMAC-SHA256) and turns every other algorithm into an opt-in. This is the recommended starting point for constrained targets.

Define	Description
WOLFCOSE_LEAN	Core-only base; all extensions become opt-in
WOLFCOSE_ENABLE_<X>	Opt in a single extension (see list below)

Extension names for WOLFCOSE_ENABLE_<X>: ES384, ES512, EDDSA, ED448, RSAPSS, MLDSA, LMS, HMAC384, HMAC512, AESCCM, CHACHA20, AESMAC, AESWRAP, ECDH_ES, SIGN (multi-signer), ENCRYPT (multi-recipient), MAC (multi-recipient), and COUNTERSIGN (RFC 9338 countersignatures).

COSE-HPKE uses its own explicit gates rather than this generic extension rule, so it never becomes enabled merely because wolfSSL provides HPKE.

An extension is compiled in when it is explicitly enabled (WOLFCOSE_ENABLE_<X>), or — in a non-lean build — when wolfSSL provides the primitive and it is not opted out with WOLFCOSE_NO_<X>. Enabling an extension wolfSSL cannot provide is a compile error. The resolved state is exposed internally as read-only WOLFCOSE_HAVE_<X> gates (e.g. WOLFCOSE_HAVE_MLDSA); sources, tests, and examples compile against those, so you set WOLFCOSE_ENABLE_*/WOLFCOSE_NO_*, not WOLFCOSE_HAVE_*.

6.3 PSA/EAT Attestation Gates

PSA/EAT support is off in every build, including a non-lean full build. The default archive does not contain wc_CoseEatPsaToken_* symbols or the PSA/EAT parser. Define the common switch, one or more profiles, and one or more COSE envelope families deliberately. The component-count limit applies to issuers and verifiers. The two claim-map limits exist only in verifier builds.

Define	Description	Default
WOLFCOSE_ENABLE_EAT_PSA_COMMON	Common PSA/EAT API and types	off
WOLFCOSE_ENABLE_EAT_PSA_CURRENT	PSA/EAT current profile	off
WOLFCOSE_ENABLE_EAT_PSA_SIGN1	Tagged Sign1 consumption	off
WOLFCOSE_ENABLE_EAT_PSA_MAC0	Tagged Mac0 consumption	off
WOLFCOSE_ENABLE_EAT_PSA_ES256	Profile claim encoding	off
WOLFCOSE_ENABLE_EAT_PSA_SIGN1_ISSUE	Tagged Sign1 issuance	off
WOLFCOSE_ENABLE_EAT_PSA_MAC0_ISSUE	Tagged Mac0 issuance	off
WOLFCOSE_ENABLE_EAT_PSA_ECDSA_IOT_PROFILE_1	Tagged ECDSA_IOT_PROFILE_1 consumption	off
WOLFCOSE_ENABLE_EAT_PSA_ID_LOOKUP_HELPER	PSA/EAT ID_LOOKUP_HELPER	off
WOLFCOSE_ENABLE_EAT_PSA_COMPONENT_INTERFACES	PSA/EAT component interfaces	off
WOLFCOSE_EAT_PSA_TFM_DERIVED	Derived: all RFC 9783 #t fm receiver algorithms/envelopes and the deprecated RFC 9053 ECDSA IDs are enabled; do not define manually	derived
WOLFCOSE_EAT_PSA_MAX_COMPONENTS	Maximum accepted software-component maps	32

Define	Description	Default
<code>WOLFCOSE_ENABLE_EAT_PSA_MAX_VERIFY_ENTRIES</code>	Only maximum claim-map entries, including extensions	64
<code>WOLFCOSE_ENABLE_EAT_PSA_MAX_VERIFY_ENTRIES_PER_COMPONENT_MAP</code>	Maximum entries in each component map	16

`WOLFCOSE_ENABLE_EAT_PSA_ISSUE` enables the claim encoder and requires the current profile. Add `WOLFCOSE_ENABLE_EAT_PSA_SIGN1_ISSUE` and/or `WOLFCOSE_ENABLE_EAT_PSA_MAC0_ISSUE` for a protected token creator. These issuer gates are independent of the Sign1 and Mac0 consumption gates, so an attester-only build does not export `wc_CoseEatPsaToken_Verify()`. It may also disable all generic verify/decrypt operations, key decoding, and `WOLFCOSE_CBOR_DECODE`; make `eat-psa-config-check` compile both issuer envelopes in exactly that encode-only configuration.

The generic operation and algorithm gates remain authoritative:

PSA/EAT path	Required generic operation	Algorithm selection
Sign1 consume	<code>WOLFCOSE_SIGN1_VERIFY</code>	<code>WOLFCOSE_ENABLE_DEPRECATED_ALGS</code> plus ES256, ES384, and/or ES512
Sign1 issue	<code>WOLFCOSE_SIGN1_SIGN</code>	<code>WOLFCOSE_ENABLE_DEPRECATED_ALGS</code> plus ES256, ES384, and/or ES512
Mac0 consume	<code>WOLFCOSE_MAC0_VERIFY</code>	HMAC256, HMAC384, and/or HMAC512
Mac0 issue	<code>WOLFCOSE_MAC0_CREATE</code>	HMAC256, HMAC384, and/or HMAC512

Use `WOLFCOSE_NO_SIGN1_SIGN`, `WOLFCOSE_NO_SIGN1_VERIFY`, `WOLFCOSE_NO_MAC0_CREATE`, or `WOLFCOSE_NO_MAC0_VERIFY` to remove a generic operation. ES256 and HMAC256 are lean-core algorithms and can be removed with `WOLFCOSE_NO_ES256` and `WOLFCOSE_NO_HMAC256`. Select ES384, ES512, HMAC384, and HMAC512 in a lean build with `WOLFCOSE_ENABLE_ES384`, `WOLFCOSE_ENABLE_ES512`, `WOLFCOSE_ENABLE_HMAC384`, and `WOLFCOSE_ENABLE_HMAC512`; use the corresponding `WOLFCOSE_NO_*` macros in a full build. A selected PSA/EAT path that has no compatible generic operation or algorithm is a compile-time configuration error.

RFC 9783 Section 5.2 requires a receiver advertising the standardized tag: `psacertified.org,2023:psa#tfm` profile to accept tagged Sign1 and Mac0 tokens with ES256/384/512 and HMAC 256/256, 384/384, and 512/512. wolfCOSE derives `WOLFCOSE_ENABLE_EAT_PSA_TFM_FULL` only when that complete set is enabled. That is a receiver-conformance gate only. An attester that enables issuance and one RFC 9783 Sign1 or Mac0 creation path can issue a correctly labelled #tfm token. A selective current-profile verifier still rejects #tfm with `WOLFCOSE_ENABLE_EAT_PSA_PROFILE` and must not advertise full receiver conformance.

See [PSA-EAT](#) for examples, profile boundaries, and security guidance.

6.4 Algorithm Gates

Per-algorithm opt-outs for the default (non-lean) build. Each also has a `WOLFCOSE_ENABLE_<X>` form for lean opt-in. ES256, AESGCM, and HMAC256 form the lean core and stay on unless explicitly opted out.

Opt-out	Algorithm	wolfSSL requirement
WOLFCOSE_NO_ES256	ECDSA P-256 (ESP256, and ES256 with deprecated IDs enabled)	HAVE_ECC, SHA-256, P-256 not disabled by NO_ECC256, and ECC_MIN_KEY_SZ <= 256
WOLFCOSE_NO_ES384	ECDSA P-384 (ESP384, and ES384 with deprecated IDs enabled)	HAVE_ECC, WOLFSSL_SHA384, HAVE_ECC384 or HAVE_ALL_CURVES, and ECC_MIN_KEY_SZ <= 384
WOLFCOSE_NO_ES512	ECDSA P-521 (ESP512, and ES512 with deprecated IDs enabled)	HAVE_ECC, WOLFSSL_SHA512, HAVE_ECC521 or HAVE_ALL_CURVES, and ECC_MIN_KEY_SZ <= 521
WOLFCOSE_NO_EDDSA	Ed25519 (and deprecated EdDSA with an Ed25519 key)	HAVE_ED25519
WOLFCOSE_NO_ED448	Ed448 (and deprecated EdDSA with an Ed448 key)	HAVE_ED448
WOLFCOSE_NO_RSAPSS	RSAPSS (PS256/384/512)	WC_RSA_PSS
WOLFCOSE_NO_MLDSA	ML-DSA (FIPS 204)	WOLFSSL_HAVE_MLDSA (wolfSSL newer than 5.9.1)
WOLFCOSE_NO_LMS	HSS/LMS (RFC 8778)	WOLFSSL_HAVE_LMS (wolfSSL 5.9.2+)
WOLFCOSE_NO_AESGCM	AES-GCM	HAVE_AESGCM
WOLFCOSE_NO_AESCCM	AES-CCM	HAVE_AESCCM
WOLFCOSE_NO_CHACHA20	ChaCha20-Poly1305	HAVE_CHACHA + HAVE_POLY1305
WOLFCOSE_NO_HMAC256	HMAC-SHA256	HMAC and SHA-256 (NO_HMAC and NO_SHA256 unset)
WOLFCOSE_NO_HMAC384	HMAC-SHA384	WOLFSSL_SHA384
WOLFCOSE_NO_HMAC512	HMAC-SHA512	WOLFSSL_SHA512
WOLFCOSE_NO_AESMAC	AES-CBC-MAC	HAVE_AES_CBC

6.4.1 Deprecated Algorithm IDs (RFC 9864)

RFC 9864 deprecates the polymorphic RFC 9053 signature IDs ES256 (-7), ES384 (-35), ES512 (-36), and EdDSA (-8) in favour of the fully-specified ESP256 (-9), ESP384 (-51), ESP512 (-52), Ed25519 (-19), and Ed448 (-53). The key format and signature encoding are unchanged, but the protected alg value is part of the signed Sig_structure, so relabelling an existing message to a replacement ID requires re-signing it. The new IDs bind the curve, so Ed25519 with an Ed448 key (or ESP256 with a P-384 key) is rejected with WOLFCOSE_E_COSE_BAD_ALG.

Define	Description
WOLFCOSE_ENABLE_DEPRECATED_ALGS	Also accept the deprecated ES256/ES384/ES512/EdDSA IDs on sign and verify. Off by default in both full and lean builds.

Without it the deprecated IDs fail with WOLFCOSE_E_COSE_BAD_ALG. Enable it to verify messages from peers that still emit the RFC 9053 IDs, including the pinned t_cose, go-cose, coset, and python-cwt interop peers, the COSE WG example vectors, and the RFC 9783 PSA attestation tokens (ES256). The CLI tool accepts the ES256 and EdDSA names only in such a build; Ed448 on the CLI always means -53.

RSA-PSS operations enforce RFC 8230's minimum 2048-bit modulus. A minimal wolfSSL RSA verify-only build must define `WOLFSSL_EXPORT_INT` so wolfCOSE can inspect the exact modulus width; builds that also enable ECC already expose the required wolfSSL integer-export API.

6.4.2 ECDSA Nonce Policy

Local ES256, ES384, and ES512 signing uses wolfSSL's configured nonce policy by default. Deterministic ECDSA is an optional hardening feature because its RFC 6979 and HMAC support increases the linked footprint and requires a specially built wolfSSL. When enabled, wolfCOSE selects SHA-256, SHA-384, or SHA-512 to match the COSE algorithm before every signature, then restores the caller's deterministic-mode and hash settings.

Define	Description	Default
<code>WOLFSSL_ENABLE_DETERMINISTIC_ECDSA</code>	Require RFC 6979 deterministic local ECDSA signing	off

The feature requires wolfSSL built with `WOLFSSL_ECDSA_DETERMINISTIC_K` or `WOLFSSL_ECDSA_DETERMINISTIC_K`. wolfSSL has no dedicated configure switch for these defines, so enable one in wolfSSL's CPPFLAGS, then enable the wolfCOSE feature when building:

```
cd wolfssl
CPPFLAGS="-DWOLFSSL_ECDSA_DETERMINISTIC_K -DWOLFSSL_NO_MALLOC" \
./configure <options>
make && sudo make install
```

```
cd ../wolfcose
make EXTRA_CFLAGS="-DWOLFSSL_ENABLE_DETERMINISTIC_ECDSA"
```

`WOLFSSL_NO_MALLOC` keeps the wolfCrypt deterministic-nonce helper on its preallocated path. It is not required to enable deterministic signing, but it preserves zero-heap operation across the wolfCOSE and wolfCrypt layers.

Verification-only builds do not need the option. A delegated signing callback chooses its own nonce policy. For a `WOLFSSL_USER_SETTINGS` test build, override the policy probe include, for example with `ECDSA_POLICY_OPTS=' -DWOLFSSL_USER_SETTINGS -Ipath/to/settings '`.

wolfCOSE also rejects local signing through wolfSSL dispatchers that do not consume the key's deterministic nonce state: crypto callbacks that always search or require callback-only ECC, STM32 PKA signing, ATECC508A/608A, Microchip TA100, Pluton, CryptoCell, Silicon Labs SE acceleration, KCAPI, SE050, and Cavium/Intel async ECC when the feature is enabled. A plain crypto-callback build may use the software fallback, but a device-bound key is rejected at runtime. The Xilinx Versal path remains permitted because wolfSSL passes its RFC 6979-derived nonce to that hardware signer.

Without `WOLFSSL_NO_MALLOC`, wolfSSL may allocate a temporary deterministic nonce object per signature. In an ES256-only x86_64 macOS link, deterministic support increased text and constants by approximately 7.1 KB. The exact cost is platform and wolfSSL-configuration dependent, which is why the feature is off by default.

6.5 Message Type Gates

6.5.1 COSE Countersignatures

Define	Description	Default
WOLFCOSE_COUNTERSIGN	Enable RFC 9338 countersignatures	Enabled
WOLFCOSE_ENABLE_COUNTERSIGN	Opt in under WOLFCOSE_LEAN	-
WOLFCOSE_NO_COUNTERSIGN	Disable countersignatures entirely	-
WOLFCOSE_COUNTERSIGN_SIGN	Enable countersignature creation	Enabled
WOLFCOSE_NO_COUNTERSIGN_SIGN	Disable countersignature creation	-
WOLFCOSE_COUNTERSIGN_VERIFY	Enable countersignature verification	Enabled
WOLFCOSE_NO_COUNTERSIGN_VERIFY	Disable countersignature verification	-

The default full build includes countersignatures. WOLFCOSE_LEAN excludes them unless WOLFCOSE_ENABLE_COUNTERSIGN is defined. Creation and verification both require CBOR encoding and decoding because each operation decodes the target message and encodes its Counter-sign-structure. Creation also requires a signing algorithm, and verification requires a verification algorithm.

6.5.2 COSE_Sign1 (Single Signer)

Define	Description	Default
WOLFCOSE_SIGN1	Enable COSE_Sign1 message type	Enabled
WOLFCOSE_NO_SIGN1	Disable COSE_Sign1 entirely	-
WOLFCOSE_SIGN1_SIGN	Enable Sign1 creation	Enabled
WOLFCOSE_NO_SIGN1_SIGN	Disable Sign1 creation	-
WOLFCOSE_SIGN1_VERIFY	Enable Sign1 verification	Enabled
WOLFCOSE_NO_SIGN1_VERIFY	Disable Sign1 verification	-

6.5.3 COSE_Encrypt0 (Symmetric Encryption)

Define	Description	Default
WOLFCOSE_ENCRYPT0	Enable COSE_Encrypt0 message type	Enabled
WOLFCOSE_NO_ENCRYPT0	Disable COSE_Encrypt0 entirely	-
WOLFCOSE_ENCRYPT0_ENCRYPT	Enable Encrypt0 creation	Enabled
WOLFCOSE_NO_ENCRYPT0_ENCRYPT	Disable Encrypt0 creation	-
WOLFCOSE_ENCRYPT0_DECRYPT	Enable Encrypt0 decryption	Enabled
WOLFCOSE_NO_ENCRYPT0_DECRYPT	Disable Encrypt0 decryption	-

6.5.4 COSE_Mac0 (Symmetric MAC)

Define	Description	Default
WOLFCOSE_MAC0	Enable COSE_Mac0 message type	Enabled
WOLFCOSE_NO_MAC0	Disable COSE_Mac0 entirely	-
WOLFCOSE_MAC0_CREATE	Enable Mac0 creation	Enabled
WOLFCOSE_NO_MAC0_CREATE	Disable Mac0 creation	-
WOLFCOSE_MAC0_VERIFY	Enable Mac0 verification	Enabled
WOLFCOSE_NO_MAC0_VERIFY	Disable Mac0 verification	-

6.5.5 COSE_Sign (Multi-Signer)

Define	Description	Default
WOLFCOSE_SIGN	Enable COSE_Sign (multi-signer)	Enabled
WOLFCOSE_NO_SIGN	Disable COSE_Sign entirely	-
WOLFCOSE_SIGN_SIGN	Enable Sign creation	Enabled
WOLFCOSE_NO_SIGN_SIGN	Disable Sign creation	-
WOLFCOSE_SIGN_VERIFY	Enable Sign verification	Enabled
WOLFCOSE_NO_SIGN_VERIFY	Disable Sign verification	-

6.5.6 COSE_Encrypt (Multi-Recipient)

Define	Description	Default
WOLFCOSE_ENCRYPT	Enable COSE_Encrypt (multi-recipient)	Enabled
WOLFCOSE_NO_ENCRYPT	Disable COSE_Encrypt entirely	-
WOLFCOSE_ENCRYPT_ENCRYPT	Enable Encrypt creation	Enabled
WOLFCOSE_NO_ENCRYPT_ENCRYPT	Disable Encrypt creation	-
WOLFCOSE_ENCRYPT_DECRYPT	Enable Encrypt decryption	Enabled
WOLFCOSE_NO_ENCRYPT_DECRYPT	Disable Encrypt decryption	-

6.5.7 COSE_Mac (Multi-Recipient)

Define	Description	Default
WOLFCOSE_MAC	Enable COSE_Mac (multi-recipient)	Enabled
WOLFCOSE_NO_MAC	Disable COSE_Mac entirely	-
WOLFCOSE_MAC_CREATE	Enable Mac creation	Enabled
WOLFCOSE_NO_MAC_CREATE	Disable Mac creation	-
WOLFCOSE_MAC_VERIFY	Enable Mac verification	Enabled
WOLFCOSE_NO_MAC_VERIFY	Disable Mac verification	-

6.6 Key Distribution Gates

Define	Description	Default
WOLFCOSE_NO_RECIPIENTS	Disable all multi-recipient support (COSE_Encrypt/COSE_Mac)	-
WOLFCOSE_NO_AESWRAP	Disable AES Key Wrap (A128KW, A192KW, A256KW)	-
WOLFCOSE_NO_ECDH_ES	Disable ECDH-ES key agreement	-
WOLFCOSE_ENABLE_AESWRAP	Opt in AES Key Wrap under WOLFCOSE_LEAN	-
WOLFCOSE_ENABLE_ECDH_ES	Opt in ECDH-ES under WOLFCOSE_LEAN	-

Resolved internally as read-only WOLFCOSE_KEY_WRAP, WOLFCOSE_ECDH, and WOLFCOSE_ECDH_WRAP gates. Requires the matching wolfSSL feature (HAVE_AES_KEYWRAP; HAVE_ECC + HAVE_HKDF for ECDH-ES) and at least one multi-recipient message type enabled. AES Key Wrap also requires wolfSSL 5.9.0 or

later; older releases used a comparison whose timing behavior depended on the compiler and XMEM-CMP configuration. Define `WOLFCOSE_NO_AESWRAP` when building the otherwise supported wolfSSL 5.8.x series.

6.7 COSE-HPKE (experimental)

COSE-HPKE implements the P0 subset of [draft-ietf-cose-hpke-26](#). It is deliberately disabled by default in both full and lean builds because the COSE binding and its IANA values are still an Internet-Draft. HPKE itself is standardized by RFC 9180, but the COSE profile is not yet an RFC.

P0 fixes every cryptographic choice: DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, and AES-128-GCM, in HPKE base mode only. There are no hidden P-384, P-521, X25519, PSK, authenticated, or alternate-AEAD paths to enable. That keeps each enabled wire operation auditable and lets targets compile out all unsupported suites.

P0 is intended for confidential provisioning, configuration, and credential delivery. It can encrypt a payload directly to one recipient, or encrypt the content once and independently protect its content-encryption key for multiple recipients. Base mode provides recipient-only confidentiality; pair the result with a COSE signature or MAC when the sender must be authenticated.

Define	Description	Default
<code>WOLFCOSE_EXPERIMENTAL</code>	Required acknowledgement for every COSE-HPKE enable; alone enables no HPKE code	off
<code>WOLFCOSE_ENABLE_HPKE_0</code>	Enable both send and receive for single-recipient integrated COSE_Encrypt0 HPKE-0	off
<code>WOLFCOSE_ENABLE_HPKE_0_ENCRYPT0</code>	Enable only integrated Encrypt0 send	off
<code>WOLFCOSE_ENABLE_HPKE_0_DECRYPT0</code>	Enable only integrated Encrypt0 receive	off
<code>WOLFCOSE_ENABLE_HPKE_0_KEY0</code>	Enable both send and receive for multi-recipient COSE_Encrypt HPKE-0-KE	off
<code>WOLFCOSE_ENABLE_HPKE_0_KEY0_ENCRYPT0</code>	Enable only multi-recipient HPKE key-encryption send	off
<code>WOLFCOSE_ENABLE_HPKE_0_KEY0_DECRYPT0</code>	Enable only multi-recipient HPKE key-encryption receive	off
<code>WOLFCOSE_NO_HPKE_0</code>	Prohibit all HPKE-0 enables; conflicts with any HPKE-0 enable	off
<code>WOLFCOSE_NO_HPKE_0_ENCRYPT0</code>	Compile out the corresponding integrated Encrypt0 direction	off
<code>WOLFCOSE_NO_HPKE_0_DECRYPT0</code>	Compile out the corresponding integrated Decrypt0 direction	off
<code>WOLFCOSE_NO_HPKE_0_KEY0_ENCRYPT0</code>	Compile out the corresponding multi-recipient key-encryption direction	off
<code>WOLFCOSE_NO_HPKE_0_KEY0_DECRYPT0</code>	Compile out the corresponding multi-recipient key-encryption direction	off

Every HPKE `ENABLE_*` macro requires `WOLFCOSE_EXPERIMENTAL`; selecting an HPKE operation without the acknowledgement is a compile error. The convenience gates respect their per-direction `NO_*` gates. Defining an individual `ENABLE_*` and its matching `NO_*` is a compile error; defining `WOLFCOSE_NO_HPKE_0` with any enable is also a compile error.

An enabled HPKE operation requires wolfSSL `HAVE_HPKE`, `HAVE_ECC` with P-256, SHA-256, and `HAVE_AESGCM`. Integrated mode also needs the matching `COSE_Encrypt0` direction. Key-encryption

mode additionally needs the matching `COSE_Encrypt` direction and recipient support; under `WOLFCOSE_LEAN`, add `WOLFCOSE_ENABLE_ENCRYPT` before enabling an HPKE-0-KE operation.

Build the wolfSSL backend with HPKE, ECC P-256, AES-GCM, SHA-256, and key generation. For example, a cryptography-only backend can use:

```
./configure --enable-cryptonly --enable-hpke --enable-ecc --enable-aesgcm \
--enable-keygen
```

The public identifiers are provisional draft values: HPKE-0 is algorithm 35, HPKE-0-KE is algorithm 46, and ek is header label -4. Applications must treat them as draft values and update when the COSE-HPKE RFC or IANA registry changes. P0 accepts only base mode: `psk_id`, a PSK, and external HPKE info are intentionally not exposed. Application external AAD is still bound through the normal COSE `Enc_structure` API argument.

For integrated mode, call `wc_CoseHpkeEncrypt0_Encrypt()` and `wc_CoseHpkeEncrypt0_Decrypt()`. It encrypts payloads directly for one recipient. For key-encryption mode, use `wc_CoseEncrypt_Encrypt()` and `wc_CoseEncrypt_Decrypt()` with every `WOLFCOSE_RECIPIENT.algId` set to `WOLFCOSE_ALG_HPKE_0_KE`; wolfCOSE generates one CEK, encrypts the content once, and HPKE-wraps that CEK separately for every recipient. HPKE base mode does not authenticate the sender, so add a COSE signature or MAC when sender authentication is required.

Draft P0 permits the HPKE `alg` header parameter to be absent. The integrated decrypt API is already pinned to HPKE-0; for key-encryption decrypt, set the selected `WOLFCOSE_RECIPIENT.algId` to `WOLFCOSE_ALG_HPKE_0_KE` so wolfCOSE can pin the omitted value safely. When an integrated or recipient HPKE `alg` is present, it must be in the protected header. wolfCOSE rejects an unprotected HPKE `alg` rather than accepting an unauthenticated key-management choice. For HPKE-0-KE, the outer `COSE_Encrypt` content algorithm may be unprotected: the HPKE `Recipient_structure` binds it as `next_layer_alg`, so a modification makes HPKE CEK recovery fail. P0 requires the 65-byte P-256 ek in the unprotected header, rejects it in the protected header, and rejects `psk_id` because PSK mode is not implemented.

```
# Single-recipient integrated HPKE: send and receive.
```

```
make EXTRA_CFLAGS="-DWOLFCOSE_EXPERIMENTAL -DWOLFCOSE_ENABLE_HPKE_0"
```

```
# Receive-only provisioning target: no HPKE sender code.
```

```
make EXTRA_CFLAGS="-DWOLFCOSE_EXPERIMENTAL -DWOLFCOSE_ENABLE_HPKE_0_DECRYPT"
```

```
# Multi-recipient provisioning server, including the lean COSE_Encrypt gate.
```

```
make EXTRA_CFLAGS="-DWOLFCOSE_LEAN -DWOLFCOSE_EXPERIMENTAL
```

```
  -DWOLFCOSE_ENABLE_ENCRYPT \
  -DWOLFCOSE_ENABLE_HPKE_0_KE_ENCRYPT"
```

Draft -26's published Section 5.2 key-encryption sample encodes a 16-byte A128GCM IV. RFC 9053 fixes the COSE AES-GCM nonce at 12 bytes, so wolfCOSE rejects that malformed sample and requires a 12-byte IV for all COSE content encryption.

6.8 Delegated Signing

Define	Description	Default
<code>WOLFCOSE_ENABLE_EXT_SIGN</code>	On in <code>wc_CoseKey_SetExtSigner()</code> , which delegates signing to a caller-supplied callback	off

Off in every build unless explicitly enabled, lean or not. Resolved internally as the read-only WOLF-COSE_EXT_SIGN gate.

Intended for keys held outside wolfCOSE — an HSM, a secure element, or a TrustZone secure partition.

What the callback receives depends on the algorithm, and getting this wrong produces a well-formed message no verifier accepts:

Algorithm	tbs holds
ES256/384/512	the digest of the Sig_structure — sign with a sign-hash primitive (psa_sign_hash, CKM_ECDSA) and return fixed-width $r \setminus \setminus \setminus s$ (RFC 9053 sec. 2.1), not a DER SEQUENCE
PS256/384/512	the digest — sign with RSASSA-PSS, MGF1 over the same SHA-2 as the algorithm, salt length equal to the digest length (RFC 8230 sec. 2)
EdDSA, Ed448, ML-DSA, HSS-LMS	the Sig_structure itself — sign it with a sign-message primitive

It returns the raw COSE signature; wolfCOSE checks the returned length against the algorithm but performs no key operation itself. No RNG is needed. HSS-LMS is stateful: the callback's own signing operation advances one-time-signature state, and the caller owns its persistence.

A delegated key needs no local *private* key, but it must declare enough for wolfCOSE to know the expected signature length: ES* and ML-DSA need nothing beyond alg; EdDSA needs kty/crv; PS* needs kty plus a local RsaKey attached via wc_CoseKey_SetRsa() for its modulus size; HSS-LMS needs an initialized LmsKey attached via wc_CoseKey_SetLms(), because its length follows the key's parameter set.

Pass a NULL callback to detach. Attaching local key material with wc_CoseKey_SetEcc() and friends detaches implicitly, so always call wc_CoseKey_SetExtSigner() last. wc_CoseKey_Decode() is rejected on a key that has a signer attached, rather than silently importing private material and signing locally with it.

For a key that has no local wolfCrypt object at all, set kty (and crv for EdDSA) on the WOLFCOSE_KEY directly before attaching the signer — there is no setter for declaring a key type without attaching one.

Two limits worth knowing before designing around this:

- It does not remove the local algorithm. WOLFCOSE_ENABLE_EXT_SIGN still requires an enabled wolfCOSE signing operation plus enough algorithm metadata to size the signature, so the verify-only profiles (WOLFCOSE_LEAN_VERIFY, WOLFCOSE_LEAN_VERIFY_MLDSA, WOLFCOSE_LEAN_VERIFY_LMS) cannot be combined with it. The wolfCrypt backend itself may lack the signing primitive: delegated HSS-LMS works against a WOLFSSL_LMS_VERIFY_ONLY wolfSSL, because the attached public LmsKey supplies the length and the callback signs. Delegating ML-DSA likewise needs local ML-DSA compiled in, which raises the WOLFCOSE_MAX_SCRATCH_SZ default to 8192 bytes and enforces a 4096-byte minimum.
- Scratch must hold the Sig_structure, which embeds the payload, and delegated signing needs at least as much again for the signature:
 - **ES*, PS*** pre-hash, so the signature reuses the Sig_structure space: `scratchSz >= max(Sig_structure, signature)`. Local ECDSA signs into a stack buffer and needs only the Sig_structure, so a small-payload ES256 case can need more scratch delegated than local.
 - **EdDSA, Ed448, ML-DSA, HSS-LMS** sign the structure in place, so the signature goes after it: `scratchSz >= Sig_structure + signature`. Delegated Ed25519 needs 64 bytes more

than the local path, Ed448 114; HSS-LMS needs its parameter-set signature length (up to 10204 bytes for the W4 predefined sets).

6.9 CBOR Layer Gates

Define	Description	Default
WOLFCOSE_CBOR_ENCODE	Enable CBOR encoding	Enabled
WOLFCOSE_NO_CBOR_ENCODE	Disable CBOR encoding	-
WOLFCOSE_CBOR_DECODE	Enable CBOR decoding	Enabled
WOLFCOSE_NO_CBOR_DECODE	Disable CBOR decoding	-

6.10 COSE_Key Gates

Define	Description	Default
WOLFCOSE_KEY_ENCODE	Enable COSE_Key encoding	Enabled
WOLFCOSE_NO_KEY_ENCODE	Disable COSE_Key encoding	-
WOLFCOSE_KEY_DECODE	Enable COSE_Key decoding	Enabled
WOLFCOSE_NO_KEY_DECODE	Disable COSE_Key decoding	-

Private RSA COSE_Key serialization requires wolfSSL 5.9.2 or later. Define WOLFCOSE_RSA_PUBLIC_ONLY to retain RSA-PSS and public-key encoding on an older wolfSSL release.

6.11 Size Configuration

Define	Description	Default
WOLFCOSE_MAX_SCRATCH_SZ	Scratch buffer size for Sig_structure/Enc_structure	512 (1024 with RSA-PSS countersigning)
WOLFCOSE_PROTECTED_HEADER_MAX	Max protected header size	64
WOLFCOSE_CBOR_MAX_DEPTH	Max CBOR nesting depth	8
WOLFCOSE_MIN_BUFFERS	Trim the working set to the minimum that fits the enabled algorithms	-

6.11.1 WOLFCOSE_MIN_BUFFERS

One define that trims the caller working set to the minimum that still fits the enabled algorithms. It tightens the CBOR parsing limits (WOLFCOSE_CBOR_MAX_DEPTH 8->6, WOLFCOSE_MAX_MAP_ITEMS 16->8) and keeps the algorithm-driven signature/scratch floors, which track the largest enabled signature algorithm:

Enabled signature algorithm	WOLFCOSE_MAX_SIG_SZ	WOLFCOSE_MAX_SCRATCH_SZ
ES256/384/512, EdDSA (Ed25519/Ed448)	132	512

Enabled signature algorithm	WOLFCOSE_MAX_SIG_SZ	WOLFCOSE_MAX_SCRATCH_SZ
RSA-PSS (PS256/384/512)	512	512 (1024 with countersigning)
ML-DSA-44/65/87	4627	8192
HSS-LMS	10240	11264

Because the floor follows the algorithm, WOLFCOSE_MIN_BUFFERS stays valid with any algorithm — ML-DSA, HSS-LMS, and RSA-PSS simply use that algorithm’s floor rather than the ECC floor. HSS-LMS uses the largest default floors because its signature size follows the key’s parameter set rather than a fixed constant: the WOLFCOSE_MAX_SIG_SZ default of 10240 and WOLFCOSE_MAX_SCRATCH_SZ default of 11264 (scratch also holds the Sig_structure) cover every W4 and W8 predefined set, the largest being L4_H10_W4 at 10204 bytes. The low-Winternitz (W1/W2) multi-level sets reach 18012 bytes and need a -D override; a key whose signature exceeds these buffers is rejected via wc_LmsKey_GetSigLen() before any signing, so no one-time state is consumed. It stays zero-heap and shrinks buffers, not stack frames. An explicit -D override of any individual limit takes precedence.

Countersignature sizing. Countersigning builds the Countersign_structure and then keeps the signature after it in scratch, so WOLFCOSE_MAX_SCRATCH_SZ must be at least WOLFCOSE_MAX_SIG_SZ + 256. The RSA-PSS default therefore rises to 1024 (a 4096-bit signature is 512 bytes) only when countersignature creation is enabled; RSA-PSS builds without it keep 512. An override below the floor is a build error.

6.12 Tuning for Size

Four levers, smallest impact last. See the [Footprint](#) page for the resulting numbers.

1. **Pick a build profile.** WOLFCOSE_LEAN is the lean ES256 core (6.8 KB glue); WOLFCOSE_LEAN_VERIFY is verify-only (5.1 KB); the ML-DSA profiles are post-quantum (see [Build Profiles](#)). One define selects a curated gate set.
2. **Drop individual features** with WOLFCOSE_NO_<X> (e.g. WOLFCOSE_NO_ENCRYPT0, WOLFCOSE_NO_SIGN, WOLFCOSE_NO_RECIPIENTS), or in a lean build add only what you need with WOLFCOSE_ENABLE_<X>.
3. **WOLFCOSE_MIN_BUFFERS** trims the caller working set to the floor for the enabled algorithms (see above).
4. **Override individual limits** if you know your payload bounds:

```
/* In your user_settings.h or build flags: */
#define WOLFCOSE_MAX_SCRATCH_SZ    256    /* default 512 */
#define WOLFCOSE_PROTECTED_HDR_MAX 32     /* default 64 */
#define WOLFCOSE_CBOR_MAX_DEPTH   4      /* default 8  */
```

Post-quantum sizing. The post-quantum signatures drive the largest floors; they auto-scale (ML-DSA-87: WOLFCOSE_MAX_SIG_SZ 4627, WOLFCOSE_MAX_SCRATCH_SZ 8192; HSS-LMS: WOLFCOSE_MAX_SIG_SZ 10240, WOLFCOSE_MAX_SCRATCH_SZ 11264, since its size follows the key’s parameter set). WOLFCOSE_LEAN_VERIFY_MLDSA is the smallest secure PQ build at 20.8 KB total, smaller than classical ES256 verify-only. Always build the application with -ffunction-sections -fdata-sections -Wl,--gc-sections so only the COSE functions you call are linked.

6.13 Tuning for Speed

The wolfCOSE layer is thin and allocation-free; end-to-end COSE_Sign1 time is dominated by the wolfCrypt backend. Tuning speed (and the backend’s own size) is a wolfSSL build concern; see the

[wolfSSL Tuning Guide](#) and the [wolfSSL Manual](#) for the assembly and math options that drive throughput (e.g. `--enable-sp-asm/WOLFSSL_SP_ARM_CORTX_M_ASM` for P-256, `--enable-aesni`) and for size (`WOLFSSL_SP_SMALL`, `WOLFSSL_AES_SMALL_TABLES`).

6.14 Build Profiles

Convenience macros that select a curated set of feature gates for a common deployment, so you do not list each `WOLFCOSE_NO_*` by hand. Each builds on `WOLFCOSE_LEAN` (core-only base) and sets each gate only if you have not already chosen it.

6.14.1 Footprint

What each profile costs (code + rodata, ES256/ML-DSA-44 COSE_Sign1, built from source with dead-code elimination). *Glue* is the wolfCOSE COSE + CBOR engine alone; *total* adds the minimal wolfCrypt backend. Full cross-library and on-device numbers are on the [Footprint](#) page.

Profile	Algorithm	wolfCOSE glue	Total + wolfCrypt
<code>WOLFCOSE_LEAN</code>	ES256 sign + verify	6.8 KB	34.6 KB
<code>WOLFCOSE_LEAN_VERIFY</code>	ES256 verify-only	5.1 KB	26.2 KB
<code>WOLFCOSE_LEAN_MLDSA</code>	ML-DSA-44 sign + verify	6.6 KB	35.8 KB
<code>WOLFCOSE_LEAN_VERIFY_MLDSA</code>	ML-DSA-44 verify-only	4.6 KB	20.8 KB

Post-quantum sign + verify is within ~1 KB of classical ES256 (35.8 vs 34.6 KB), and PQ verify-only is actually *smaller* than classical ES256 verify-only (20.8 vs 26.2 KB): ML-DSA skips the DER signature conversion ECDSA needs. Full numbers (desktop, on-device, and speed) are on the [Footprint](#) page.

6.14.2 WOLFCOSE_LEAN_VERIFY — minimal verify-only

The smallest secure on-device profile: COSE_Sign1 verification only, the common case where a device verifies signed firmware or attestation while signing happens off-device on a server or HSM. It implies `WOLFCOSE_LEAN` plus `WOLFCOSE_NO_SIGN1_SIGN` (removing signing and, transitively, the RNG), `WOLFCOSE_NO_ENCRYPT0`, `WOLFCOSE_NO_MAC0`, `WOLFCOSE_NO_MAC0_CREATE`, `WOLFCOSE_NO_KEY_ENCODE`, and `WOLFCOSE_NO_KEY_DECODE`. An explicit `WOLFCOSE_ENABLE_EAT_PSA_MAC0` selection retains Mac0 verification but not creation, so a verify-only full RFC 9783 #tfm receiver can opt in. Full RFC 9052 verification stays: header decode, crit enforcement, duplicate-label detection, and the Sig_structure rebuild. Sign1 verify must stay enabled; the build errors out if it is also disabled.

```
make lean-verify      # builds + runs examples/sign1_verify_lean.c with the
  ↪ profile
# or directly:
cc -DWOLFCOSE_LEAN_VERIFY ... src/*.c
```

6.14.3 WOLFCOSE_LEAN_MLDSA — lean post-quantum sign + verify

A lean ML-DSA-only (FIPS 204) COSE_Sign1 **sign and verify** profile. It implies `WOLFCOSE_LEAN` plus `WOLFCOSE_ENABLE_MLDSA` (ML-DSA is an extension, off under `WOLFCOSE_LEAN`), `WOLFCOSE_NO_ES256` (PQ-only, so the ECDSA path compiles out), `WOLFCOSE_NO_ENCRYPT0`, `WOLFCOSE_NO_MAC0`, `WOLFCOSE_NO_KEY_ENCODE`, and `WOLFCOSE_NO_KEY_DECODE`, keeping **both** Sign1 sign and verify. Pair it with a wolfCrypt backend built with ML-DSA (`--enable-dilithium`).

```
make mldsa-demo      # builds + runs examples/sign1_mldsa.c (sign + verify)
# or directly:
cc -DWOLFCOSE_LEAN_MLDSA ... src/*.c
```

6.14.4 WOLFCOSE_LEAN_VERIFY_MLDSA — minimal post-quantum verify-only

The smallest secure on-device PQ build: ML-DSA COSE_Sign1 **verify only**. It implies WOLFCOSE_LEAN_MLDSA plus WOLFCOSE_NO_SIGN1_SIGN, so signing and the RNG it needs are not compiled in, while full RFC 9052 verification stays intact. Pair with a wolfCrypt build that enables ML-DSA **verify** only (e.g. WOLFSSL_DILITHIUM_VERIFY_ONLY).

```
make mldsa-verify    # builds + runs examples/sign1_verify_mldsa.c with the
↳ profile
# or directly:
cc -DWOLFCOSE_LEAN_VERIFY_MLDSA ... src/*.c
```

6.14.5 WOLFCOSE_LEAN_LMS — lean stateful hash-based sign + verify

A lean HSS/LMS-only (RFC 8778, SP 800-208) COSE_Sign1 **sign and verify** profile with the same implied gate set as WOLFCOSE_LEAN_MLDSA (WOLFCOSE_LEAN, WOLFCOSE_ENABLE_LMS, WOLFCOSE_NO_ES256, no Encrypt0/Mac0/key codec). LMS signing is stateful: signing advances one-time-signature state inside wolfCrypt, and the caller installs the wolfCrypt private-key read/write callbacks that persist it. wolfCOSE stores no state itself and defines no persistence policy. Pair with a wolfCrypt backend built with LMS (-enable-lms).

```
make lms-demo        # builds + runs examples/sign1_lms.c with the profile
```

6.14.6 WOLFCOSE_LEAN_VERIFY_LMS — minimal hash-based verify-only

HSS/LMS COSE_Sign1 **verify only**: implies WOLFCOSE_LEAN_LMS plus WOLFCOSE_NO_SIGN1_SIGN. Verification is hash operations only (no big-integer or lattice math), so this pairs naturally with a wolfCrypt LMS verify-only build (WOLFSSL_LMS_VERIFY_ONLY), the CNSA 2.0 firmware-verification shape. The signature length follows the key's levels/height/Winternitz parameter set, so the WOLFCOSE_MAX_SIG_SZ default rises to 10240 and WOLFCOSE_MAX_SCRATCH_SZ to 11264, covering every W4/W8 predefined set; the low-Winternitz (W1/W2) multi-level sets need a -D override.

```
make lms-verify      # builds + runs examples/sign1_verify_lms.c with the
↳ profile
```

6.15 Example Build Configurations

6.15.1 Sign-Only Build (Minimal)

```
make CFLAGS="-DWOLFCOSE_NO_ENCRYPT0 -DWOLFCOSE_NO_MAC0 -DWOLFCOSE_NO_ENCRYPT
↳ -DWOLFCOSE_NO_MAC"
```

6.15.2 Verify-Only Build

```
make CFLAGS="-DWOLFCOSE_NO_SIGN1_SIGN -DWOLFCOSE_NO_ENCRYPT0_ENCRYPT
↳ -DWOLFCOSE_NO_MAC0_CREATE"
```

6.15.3 Sign1-Only Build (Smallest)

```
make CFLAGS="-DWOLFCOSE_NO_ENCRYPT0 -DWOLFCOSE_NO_MAC0 -DWOLFCOSE_NO_SIGN
↳ -DWOLFCOSE_NO_ENCRYPT -DWOLFCOSE_NO_MAC"
```

6.15.4 No Multi-Recipient Support

```
make CFLAGS="-DWOLFCOSE_NO_RECIPIENTS"
```

6.16 wolfSSL Dependencies

wolfCOSE requires these wolfSSL features for full functionality:

wolfSSL Define	wolfCOSE Feature
HAVE_ECC	ECDSA signing (ES256/ES384/ES512), ECDH key agreement
WOLFSSL_ECDSA_DETERMINISTIC_K or WOLFSSL_ECDSA_DETERMINISTIC_K_VARIANT	Optional deterministic local ECDSA signing
HAVE_ED25519	EdDSA signing (Ed25519)
HAVE_ED448	EdDSA signing (Ed448)
WOLFSSL_HAVE_MLDSA	ML-DSA post-quantum signing
WC_RSA_PSS	RSA-PSS signing (PS256/PS384/PS512)
HAVE_AESGCM	AES-GCM encryption
HAVE_AESCCM	AES-CCM encryption
HAVE_CHACHA && HAVE_POLY1305	ChaCha20-Poly1305 encryption
HAVE_AES_CBC	AES-CBC-MAC
NO_HMAC (NOT defined)	HMAC algorithms
WOLFSSL_SHA384	SHA-384 for ES384, HMAC-384
WOLFSSL_SHA512	SHA-512 for ES512, HMAC-512
HAVE_AES_KEYWRAP	AES Key Wrap distribution
HAVE_HKDF	ECDH-ES key derivation
HAVE_HPKE + HAVE_ECC P-256 + SHA-256 + HAVE_AESGCM	Experimental COSE-HPKE PO

6.17 Test and Example Gates

6.17.1 Comprehensive Test Gates

Each comprehensive test file can be disabled:

Define	Description
WOLFCOSE_NO_EXAMPLE_SIGN_ALL	Disable sign_all.c
WOLFCOSE_NO_EXAMPLE_ENCRYPT_ALL	Disable encrypt_all.c
WOLFCOSE_NO_EXAMPLE_MAC_ALL	Disable mac_all.c
WOLFCOSE_NO_EXAMPLE_ERRORS_ALL	Disable errors_all.c

Sub-gates within tests:

Define	Description
WOLFCOSE_NO_SIGN_ALL_ES256	Skip ES256 tests in sign_all
WOLFCOSE_NO_SIGN_ALL_MULTI	Skip multi-signer tests

Define	Description
WOLFCOSE_NO_ENCRYPT_ALL_A128GCM	Skip A128GCM tests
WOLFCOSE_NO_MAC_ALL_HMAC256	Skip HMAC-256 tests

6.17.2 Scenario Example Gates

Define	Description
WOLFCOSE_NO_EXAMPLE_FIRMWARE_UPDATE	Disable firmware_update.c
WOLFCOSE_NO_EXAMPLE_MULTI_PARTY	Disable multi_party_approval.c
WOLFCOSE_NO_EXAMPLE_IOT_FLEET	Disable iot_fleet_config.c
WOLFCOSE_NO_EXAMPLE_SENSOR_ATTEST	Disable sensor_attestation.c
WOLFCOSE_NO_EXAMPLE_GROUP_BROADCAST	Disable group_broadcast_mac.c

6.18 See Also

- **Getting Started:** Build instructions
- **Algorithms:** Supported algorithms with guards
- **Testing:** Test configuration

7 Project Structure

This page describes the source code layout and file organization of wolfCOSE.

7.1 Directory Layout

```
wolfCOSE/|—
  include/|—
    wolfcose/|—
      wolfcose.h          # Public API (types, constants, all functions)|—
      visibility.h        # WOLFCOSE_API export macros|—
  src/|—
    wolfcose_cbor.c       # CBOR encoder/decoder (no wolfCrypt dependency)
    |—
    wolfcose_util.c       # Zeroization, rollback, constant-time compare|—
    wolfcose_alg.c        # Algorithm dispatch (hash/AEAD/HMAC/RSA-PSS)|—
    wolfcose_ecc.c        # ECC DER <-> raw r||s conversion|—
    wolfcose_hdr.c        # Protected/unprotected header codec|—
    wolfcose_key.c        # COSE_Key encode/decode + key attachment|—
    wolfcose_struct.c     # Sig/MAC/Enc structure builders|—
    wolfcose_recipient.c  # Key wrap + ECDH-ES key agreement|—
    wolfcose_sign1.c      # COSE_Sign1|—
    wolfcose_sign.c       # COSE_Sign (multi-signer)|—
    wolfcose_encrypt0.c   # COSE_Encrypt0|—
    wolfcose_mac0.c       # COSE_Mac0|—
    wolfcose_encrypt.c    # COSE_Encrypt (multi-recipient)|—
    wolfcose_mac.c        # COSE_Mac (multi-recipient)|—
    wolfcose_internal.h   # Internal shared declarations|—
  tests/|—
    test_cbor.c           # CBOR unit tests|—
    test_cose.c           # COSE unit tests|—
    test_interop.c        # Interoperability tests|—
    test_main.c           # Test harness|—
    force_failure.c       # Failure injection|—
    force_failure.h       # Failure injection API|—
    vectors/              # Test vectors|—
  tools/|—
    wolfcose_tool.c       # CLI tool|—
  examples/|—
    sign1_demo.c          # Sign1 algorithm demo|—
    encrypt0_demo.c       # Encrypt0 algorithm demo|—
    mac0_demo.c           # Mac0 algorithm demo|—
    lifecycle_demo.c      # Edge-to-cloud demo|—
    comprehensive/       # Algorithm matrix tests||—
      sign_all.c||—
      encrypt_all.c||—
```

```

    mac_all.c | | └─
    errors_all.c | └─
    scenarios/          # Real-world scenarios | └─
        firmware_update.c | └─
        multi_party_approval.c | └─
        iot_fleet_config.c | └─
        sensor_attestation.c | └─
        group_broadcast_mac.c | └─
    Makefile └─
    README.md

```

7.2 Core Library Files

7.2.1 include/wolfcose/wolfcose.h

The public API header. Contains: - All type definitions (WOLFCOSE_KEY, WOLFCOSE_HDR, etc.) - Algorithm constants (WOLFCOSE_ALG_ESP256, etc.) - Key type and curve constants - COSE tag values - Error codes - All function declarations

7.2.2 include/wolfcose/visibility.h

Export macros for shared library builds: - WOLFCOSE_API for public functions - Platform-specific visibility attributes

7.2.3 COSE message modules (src/wolfcose_*.c)

The COSE implementation (RFC 9052/9053) is split into cohesive modules, one per functional area:

File	Contents
wolfcose_util.c	Secure zeroization, key-import rollback, constant-time compare, RFC 9052 context strings
wolfcose_alg.c	Algorithm dispatch: hash/signature/curve mapping, AEAD and HMAC parameters, RSA-PSS key checks
wolfcose_ecc.c	ECC DER <-> raw r
wolfcose_hdr.c	Protected/unprotected header encode and decode
wolfcose_key.c	COSE_Key encode/decode and wolfCrypt key attachment
wolfcose_struct.c	Unified Sig_structure/MAC_structure/Enc_structure builders
wolfcose_recipient.c	AES key wrap and ECDH-ES key agreement for multi-recipient messages
wolfcose_sign1.c	COSE_Sign1 sign and verify
wolfcose_sign.c	COSE_Sign multi-signer sign and verify
wolfcose_encrypt0.c	COSE_Encrypt0 encrypt and decrypt
wolfcose_mac0.c	COSE_Mac0 create and verify
wolfcose_encrypt.c	COSE_Encrypt multi-recipient encrypt and decrypt
wolfcose_mac.c	COSE_Mac multi-recipient create and verify

These files depend on wolfCrypt for all cryptographic operations.

7.2.4 src/wolfcose_cbor.c

CBOR encoder/decoder (RFC 8949). Contains: - Major type encoding/decoding - Integer, byte string, text string handling - Array and map handling - Tag encoding/decoding - Skip functionality for unknown items - Zero-copy decoding (pointers into input buffer)

This file has no wolfCrypt dependency. CBOR-only projects can link just `wolfcose_cbor.o`.

7.2.5 src/wolfcose_internal.h

Internal helpers not exposed in public API: - Big-endian macros for CBOR encoding - Protected header codec - AEAD algorithm dispatch tables - Internal constants

7.3 Build Outputs

File	Description
<code>libwolfcose.a</code>	Static library
<code>libwolfcose.so</code>	Shared library (with <code>make shared</code>)
<code>tests/test_wolfcose</code>	Test executable
<code>tools/wolfcose_tool</code>	CLI tool executable

The core library is one object file per source module. `wolfcose_cbor.o` is the only object with no wolfCrypt dependency; with `-ffunction-sections` and `--gc-sections` the linker keeps only what the application uses.

7.4 Test Files**7.4.1 tests/test_cbor.c**

CBOR unit tests covering: - RFC 8949 Appendix A encoding vectors - Decoding tests for all major types - Round-trip encode/decode tests - Nested structure tests - Skip functionality tests - Error case handling

7.4.2 tests/test_cose.c

COSE unit tests covering: - All six message types (Sign1, Sign, Encrypt0, Encrypt, Mac0, Mac) - All supported algorithms - Detached payload mode - External AAD - Multi-signer and multi-recipient messages - Error handling (tampered messages, wrong keys, etc.)

7.4.3 tests/test_interop.c

Interoperability tests with: - COSE Working Group test vectors - RFC examples - Cross-implementation compatibility

7.4.4 tests/test_main.c

Test harness that: - Initializes wolfCrypt - Runs all test suites - Reports pass/fail with CI-compatible exit codes

7.4.5 tests/force_failure.c / force_failure.h

Failure injection system for coverage testing. See [Testing](#) for details.

7.5 CLI Tool

7.5.1 tools/wolfcose_tool.c

Command-line interface providing: - Key generation for all algorithm types - Sign and verify operations - Encrypt and decrypt operations - MAC create and verify operations - COSE message inspection (info command) - Self-test functionality

Supported commands:

```
keygen    - Generate a new key
sign      - Create COSE_Sign1 message
verify    - Verify COSE_Sign1 message
enc       - Create COSE_Encrypt0 message
dec       - Decrypt COSE_Encrypt0 message
mac       - Create COSE_Mac0 message
macverify - Verify COSE_Mac0 message
info      - Display COSE message structure
test      - Run self-tests
```

7.6 Examples

7.6.1 Basic Demos

File	Description
sign1_demo.c	Demonstrates all Sign1 algorithms (ESP256, ESP384, ESP512, Ed25519, PS256/384/512, ML-DSA)
encrypt0_demo.c	Demonstrates all Encrypt0 algorithms (AES-GCM, ChaCha20, AES-CCM)
mac0_demo.c	Demonstrates all Mac0 algorithms (HMAC, AES-MAC)
lifecycle_demo.c	Full edge-to-cloud workflow with 11 algorithms

7.6.2 Comprehensive Tests (examples/comprehensive/)

File	Tests
sign_all.c	Sign1 + multi-signer matrix (~61 tests)
encrypt_all.c	Encrypt0 + multi-recipient matrix (~23 tests)
mac_all.c	Mac0 + multi-recipient matrix (~32 tests)
errors_all.c	Error handling and edge cases (~19 tests)

7.6.3 Scenario Examples (examples/scenarios/)

File	Scenario
<code>firmware_update.c</code>	Post-quantum ML-DSA firmware signing with detached payload
<code>multi_party_approval.c</code>	Dual-control firmware approval (ESP256 + ESP384)
<code>iot_fleet_config.c</code>	Encrypted config push to IoT device fleet with multiple recipients
<code>sensor_attestation.c</code>	EAT-style attestation with replay protection via external AAD
<code>group_broadcast_mac.c</code>	Authenticated broadcast to multiple subscribers

7.7 Integration Notes

7.7.1 Minimal Integration

For a minimal COSE integration, you need: 1. `include/wolfcose/wolfcose.h` 2. `include/wolfcose/visibility.h` 3. Every `src/*.c` file 4. `src/wolfcose_internal.h`

Unused message types compile out via the `WOLFCOSE_NO_*` feature macros, and `-ffunction-sections/--gc-sections` discards unreferenced code at link time.

7.7.2 CBOR-Only Integration

If you only need CBOR (no COSE): 1. `include/wolfcose/wolfcose.h` (for CBOR types) 2. `src/-wolfcose_cbor.c`

No wolfCrypt dependency required for CBOR-only.

7.7.3 Production Builds

Do NOT include in production firmware: - `tools/` directory - `examples/` directory - `tests/` directory

7.8 See Also

- [Getting Started](#): Build instructions
- [API Reference](#): Function documentation
- [Testing](#): Test organization

8 PSA Attestation and EAT Tokens

wolfCOSE can consume and create the PSA Token profile of EAT specified by [RFC 9783](#). It is an explicit opt-in: a normal `libwolfcose.a` has neither the PSA/EAT parser nor its public API symbols.

The API is designed for a verifier receiving the bytes returned by `psa_initial_attest_get_token()`, a device onboarding service, a gateway, or a remote attestation component such as wolfTrust. All storage remains owned by the caller. Decoded claim spans borrow the authenticated input token.

8.1 Select only what is needed

`WOLFCOSE_ENABLE_EAT_PSA` enables the common API. Select at least one profile and at least one consume or issue operation after it.

Define	Effect	Default
<code>WOLFCOSE_ENABLE_EAT_PSA</code>	Common PSA/EAT API and types	off
<code>WOLFCOSE_ENABLE_EAT_PSA_CURRENT</code>	RFC 9783 TF-M profile, tag:psacertified.org,2023:psa#tfm	off
<code>WOLFCOSE_ENABLE_EAT_PSA_SIGN1</code>	Tagged COSE_Sign1 consumption	off
<code>WOLFCOSE_ENABLE_EAT_PSA_MAC0</code>	Tagged COSE_Mac0 consumption	off
<code>WOLFCOSE_ENABLE_EAT_PSA_ISSUE</code>	Current-profile claim encoding	off
<code>WOLFCOSE_ENABLE_EAT_PSA_SIGN1_TOKEN</code>	Tagged COSE_Sign1 token creation	off
<code>WOLFCOSE_ENABLE_EAT_PSA_MAC0_TOKEN</code>	Tagged COSE_Mac0 token creation	off
<code>WOLFCOSE_ENABLE_EAT_PSA_LEGACY</code>	Legacy PSA_IOT_PROFILE_1 consumption	off
<code>WOLFCOSE_ENABLE_EAT_PSA_VERIFY_BY_ID</code>	<code>psa_verify_by_id()</code> key lookup helper	off
<code>WOLFCOSE_ENABLE_EAT_PSA_FOREACH_COMPONENT</code>	<code>psa_for_each_component()</code> callback helper	off
<code>WOLFCOSE_EAT_PSA_MAX_COMPONENTS</code>	Maximum accepted software-component maps	32
<code>WOLFCOSE_EAT_PSA_MAX_CLAIMS</code>	Verifier-only maximum current/legacy claim-map entries	64
<code>WOLFCOSE_EAT_PSA_MAX_COMPONENT_CLAIMS</code>	Verifier-only maximum entries in each component map	16

The generic wolfCOSE algorithm gates remain authoritative. Select ES384, ES512, HMAC384, and HMAC512 with their normal `WOLFCOSE_ENABLE_*` macros in a lean build, or remove any supported algorithm with its `WOLFCOSE_NO_*` macro. Sign1 consumption and issuance require `WOLFCOSE_ENABLE_DEPRECATED_ALGS` because RFC 9783 Table 4 uses the RFC 9053 ES256/384/512 IDs. The PSA/EAT code has no algorithm fallback: a disabled algorithm is rejected before claims are used.

RFC 9783 Sign1 tokens use the RFC 9053 ES256/ES384/ES512 IDs. Define `WOLFCOSE_ENABLE_DEPRECATED_ALGS` for a Sign1 receiver or issuer; it is also required for the derived full `#tfm` receiver capability. Without

it, those token algorithms are rejected.

The consume and issue directions are independent. An attester can select the common issue gate and one issue envelope without selecting either consumption gate. Such a build contains no PSA/EAT verifier symbol. A claim-encoder-only build can omit both issue-envelope gates. Conversely, a verifier can omit the common issue gate and every issuance API.

Minimal current Sign1 verifier:

```
-DWOLFCOSE_LEAN_VERIFY
-DWOLFCOSE_ENABLE_DEPRECATED_ALGS
-DWOLFCOSE_ENABLE_EAT_PSA
-DWOLFCOSE_ENABLE_EAT_PSA_CURRENT
-DWOLFCOSE_ENABLE_EAT_PSA_SIGN1
```

Current Mac0 verifier:

```
-DWOLFCOSE_LEAN
-DWOLFCOSE_ENABLE_EAT_PSA
-DWOLFCOSE_ENABLE_EAT_PSA_CURRENT
-DWOLFCOSE_ENABLE_EAT_PSA_MAC0
```

Current ES256 Sign1 issuer without the PSA/EAT verifier:

```
-DWOLFCOSE_LEAN
-DWOLFCOSE_NO_SIGN1_VERIFY
-DWOLFCOSE_NO_ENCRYPT0_DECRYPT
-DWOLFCOSE_NO_MAC0_VERIFY
-DWOLFCOSE_NO_KEY_DECODE
-DWOLFCOSE_NO_CBOR_DECODE
-DWOLFCOSE_ENABLE_DEPRECATED_ALGS
-DWOLFCOSE_ENABLE_EAT_PSA
-DWOLFCOSE_ENABLE_EAT_PSA_CURRENT
-DWOLFCOSE_ENABLE_EAT_PSA_ISSUE
-DWOLFCOSE_ENABLE_EAT_PSA_SIGN1_ISSUE
```

Replace the last gate with `WOLFCOSE_ENABLE_EAT_PSA_MAC0_ISSUE` for an HMAC256 Mac0 issuer. The same decode opt-outs apply. The algorithm gates can then replace ES256 or HMAC256 independently as described in [Macros](#).

These selective examples are useful COSE building blocks, but they are not RFC 9783 #tfm receivers: Section 5.2 requires a #tfm receiver to accept both envelope types and ES256/384/512 plus HMAC 256/256, 384/384, and 512/512. wolfCOSE therefore derives `WOLFCOSE_EAT_PSA_TFM_FULL` only for that complete configuration. Without it, the PSA/EAT verifier rejects the standard tag:psacertified.org,2023:psa#tfm value with `WOLFCOSE_E_EAT_PSA_PROFILE`. This receiver requirement does not constrain an attester: RFC 9783 permits it to issue #tfm with one enabled Table 4 Sign1 or Mac0 protection algorithm. A reduced verifier must not advertise full #tfm receiver conformance.

A feature-complete test profile is available through `make eat-psa-test`. It also enables current issuance, legacy consumption, UEID resolution, and component iteration. `make eat-psa-config-check` proves the default archive contains no PSA/EAT symbols and checks minimal, partial, and invalid configurations.

`make eat-psa-profile-test` then runs current Sign1-only, current Mac0-only, legacy Sign1-only, and legacy Mac0-only binaries. It verifies that the two partial current attesters can issue and generically authenticate #tfm with their enabled envelope, while their incomplete PSA/EAT receivers reject it. It also verifies that disabled standardized claim namespaces are not silently skipped and that a disabled envelope returns `WOLFCOSE_E_UNSUPPORTED` before cryptographic work.

The claim-map limits are intentionally separate from `WOLFCOSE_MAX_MAP_ITEMS`: EAT receivers accept extension claims, but must still bound input work while checking that every standardized and extension key is unique. Set either limit larger for an expected extension-rich token; the values must be at least 10 and 5 respectively.

`make eat-psa-claim-limits-test` recompiles a dedicated test with the minimum valid ceilings (10 top-level claims and 5 component-map claims). It accepts the exact boundary and rejects each respective +1 map before iterating its entries.

`WOLFCOSE_EAT_PSA_TFM_FULL` is derived only when a build has current-profile Sign1 and Mac0 verification plus ES256, ES384, ES512, HMAC256, HMAC384, and HMAC512. That is the set RFC 9783 requires a TF-M profile receiver to support; applications do not define this macro themselves; an externally supplied definition is a configuration error.

8.2 API

Include the API only in an enabled build:

```
#include <wolfcose/eat_psa.h>
```

`wc_CoseEatPsaToken_Verify()` authenticates a tagged, attached COSE token with a caller-owned `WOLFCOSE_KEY`, validates the selected PSA profile's required claim shape, and compares the authenticated nonce. It fills a `WOLFCOSE_EAT_PSA_TOKEN` only on success.

```
WOLFCOSE_EAT_PSA_TOKEN token;
int ret = wc_CoseEatPsaToken_Verify(&iak_key, token_bytes, token_size,
    challenge, challenge_size, scratch, sizeof(scratch), &token);
if (ret == WOLFCOSE_SUCCESS) {
    /* Appraise token.clientId, token.lifecycle, implementationId, and claims.
    ↪ */
}
```

`expectedNonce` is mandatory and must be 32, 48, or 64 bytes. The output spans, including nonce, `uuid`, `implementationId`, and components, point into `token_bytes`; retain that buffer unchanged until appraisal is complete.

8.2.1 PSA client handoff

`wolfCOSE` deliberately does not link against a PSA client library. A PSA client obtains opaque bytes from the standard API, then passes them directly to the verifier with a provisioned IAk key:

```
#include <psa/initial_attestation.h>
```

```
uint8_t token_bytes[1024u]; /* Select a capacity for the PSA platform. */
size_t token_size = 0u;
psa_status_t psa_ret;
```

```
psa_ret = psa_initial_attest_get_token_size(sizeof(challenge), &token_size);
if ((psa_ret == PSA_SUCCESS) && (token_size <= sizeof(token_bytes))) {
    psa_ret = psa_initial_attest_get_token(challenge, sizeof(challenge),
        token_bytes, sizeof(token_bytes), &token_size);
}
if (psa_ret == PSA_SUCCESS) {
    ret = wc_CoseEatPsaToken_Verify(&provisioned_iak, token_bytes, token_size,
        challenge, sizeof(challenge), scratch, sizeof(scratch), &token);
}
```

For a fleet verifier, enable the UEID resolver and map the untrusted UEID to a candidate provisioned IAK. The token is authenticated only after that lookup. This keeps the wolfCOSE library portable while allowing a PSA client such as wolfTrust to use its normal `psa_initial_attest_get_token()` API.

If the attestation key is selected by device identity, enable `WOLFCOSE_ENABLE_EAT_PSA_UEID_RESOLVER` and use `wc_CoseEatPsaToken_VerifyByUeid()`. The resolver sees an untrusted decoded UEID only to choose a candidate key. wolfCOSE then verifies the original token before returning any claims, so the resolver must not use that preliminary UEID for authorization.

If software-component appraisal is needed, enable `WOLFCOSE_ENABLE_EAT_PSA_COMPONENT_ITERATOR` and call `wc_CoseEatPsaToken_ForEachComponent()` after successful verification. It decodes one component at a time from the authenticated token, avoiding heap storage for a component list. The callback's component structure is valid only for the duration of that callback. Its span data borrows from the verified token input and remains valid while that input remains unchanged.

For a current-profile Sign1 issuer, enable `WOLFCOSE_ENABLE_EAT_PSA_ISSUE` and `WOLFCOSE_ENABLE_EAT_PSA_SIGN` then use:

```
wc_CoseEatPsaToken_CreateSign1(&issuer_key, WOLFCOSE_ALG_ES256, &claims,
    claims_buf, sizeof(claims_buf), scratch, sizeof(scratch), token,
    sizeof(token), &token_len, rng);
```

`wc_CoseEatPsaToken_CreateMac0()` is available only when the common issue and Mac0 issue gates are selected. Issuance always emits the current RFC 9783 profile; legacy is deliberately consume-only. An attester needs only one enabled RFC 9783 Table 4 Sign1 or Mac0 creation path to emit the standardized #t fm identifier. `WOLFCOSE_EAT_PSA_TFM_FULL` is deliberately separate: it proves a receiver has every mandatory envelope and algorithm, so a selective verifier continues to reject #t fm with `WOLFCOSE_E_EAT_PSA_PROFILE`.

The issuer's `claimsBuf`, `scratch`, and output buffer ranges must be pairwise disjoint. Either creation function rejects exact or partial overlap with `WOLFCOSE_E_INVALID_ARG` before encoding claims and clears the output range. `claimsBuf` is the output of `wc_CoseEatPsaToken_EncodeClaims()`, so it must also be disjoint from the claims structure, component array, and every nonempty claim or component span. The direct encoder enforces the same rule for its output buffer; in-place claim encoding is not supported.

The Sign1 issuer accepts an external signing key installed with `wc_CoseKey_SetExtSigner()`. The callback receives the algorithm's digest of the COSE `Sig_structure`, which maps directly to a PSA `psa_sign_hash()` implementation. Pass NULL for `rng` when the external signer owns all randomness. This is the intended wolfTrust and secure-element integration path: wolfCOSE owns the standard claim encoding and COSE envelope, while the IAK remains inside the PSA service, HSM, or secure partition.

For wolfTrust, the migration path is to have the Initial Attestation service populate `WOLFCOSE_EAT_PSA_CLAIMS`, install its existing secure-partition signing callback, and call `wc_CoseEatPsaToken_CreateSign1()`. The returned bytes can then be consumed by `wc_CoseEatPsaToken_Verify()` or the UEID resolver without a second private claim parser.

`make psa-eat-demo` runs the same application flow without a PSA service. It measures a sample secure-partition image, issues a current-profile Sign1 token, selects a provisioned public IAK by UEID, rejects a mismatched challenge, authenticates the real challenge, and applies an onboarding policy to the lifecycle, device IDs, signer, version, and software measurement. The example is in `examples/psa-eat-demo.c`.

The pre-RFC profile used by older wolfTrust trees (<http://arm.com/psa/2.0.0>) is intentionally not treated as RFC 9783. A wolfTrust migration replaces its manual claim encoder with these current-profile claims, uses the RFC 9783 tag: `psacertified.org,2023:psa#t fm` profile value, and retains its existing external IAK signer. That gives the PSA service a standards-conformant token without moving private IAK material out of its secure partition.

8.3 Profile behavior

Capability	Current RFC 9783 TF-M	Legacy PSA IoT Profile 1
Profile identifier	tag:psacertified.org,2023:PSA-TF-M-PROFILE_1	PSA-IOT-PROFILE_1
Consume	WOLFCOSE_ENABLE_EAT_PSA_CURRENT	WOLFCOSE_ENABLE_EAT_PSA_LEGACY
Issue	yes, with one enabled RFC 9783 Sign1 or Mac0 creation path	no
Required nonce	32, 48, or 64 bytes	32, 48, or 64 bytes
UEID	33 bytes, first byte 0x01	same
Boot seed	optional, 8 to 32 bytes when present	required, 32 bytes
Certification reference	current 13-digit plus version form	old 13-digit form
Components	one or more measurement components	components or explicit no-measurements form

For the current profile, wolfCOSE requires nonce, UEID, profile, nonzero client ID, lifecycle, 32-byte implementation ID, and at least one software component. Each component needs a 32, 48, or 64 byte measurement value and signer ID. Optional values and unknown EAT claims are retained or skipped as appropriate without weakening the required-claim checks. Both standardized current and legacy label namespaces are always recognized: a disabled or mixed namespace is rejected as WOLF-COSE_E_EAT_PSA_PROFILE, not treated as an extension.

The RFC 9783 0x00xx lifecycle range is structurally valid but represents an unknown state. Verification retains it; an appraisal policy should normally reject it for a trust decision.

8.4 COSE and trust boundary

RFC 9783 specifies COSE tag 18 (COSE_Sign1) and tag 17 (COSE_Mac0) for this profile. wolfCOSE accepts only an explicitly tagged, attached token and requires its alg parameter in the protected header because this API does not use externally supplied authenticated data. It rejects untagged messages, CWT tag 61 wrappers, detached payloads, indefinite-length CBOR, unsupported PSA algorithms, duplicate selected claims, and trailing bytes.

RFC 9783 requires variation-tolerant reception of definite-length CBOR. PSA/EAT verification therefore accepts non-preferred but well-formed definite encodings. Ordinary wolfCOSE decoding remains strict by default. Unknown non-critical COSE header extensions may use integer or text labels; text labels are tracked for duplicates across protected and unprotected buckets, but are not aliases for registered numeric headers. A text extension listed in crit is rejected because this verifier does not implement it.

The raw-key API rejects x5chain headers with WOLFCOSE_E_UNSUPPORTED. A certificate chain is not proof of a trusted IAK until an application has performed path building, trust-anchor selection, validity checking, and revocation policy. Do that in a certificate-aware layer, then call the raw-key API with the validated public key.

Successful verification is not complete attestation appraisal. The relying party must apply its own policy to the client ID, lifecycle, UEID, implementation ID, signer IDs, component measurements, version data, and any verification-service indicator. It must also retain replay state beyond the nonce exchange when its protocol requires it.

Mac0 is available because RFC 9783 defines it, but it uses a shared secret. Use it only where shared-key provisioning and verifier identity fit the threat model; Sign1 is normally the better choice for remotely provisioned device attestation.

8.5 Validation and interoperability

`make eat-psa-test` includes the RFC 9783 Appendix A Sign1 and Mac0 vectors, which the RFC identifies as generated by TF-M's `iat-verifier`, plus current ES256/384/512 and HMAC256/384/512 round trips, legacy cases, nonce checks, and malformed-token negatives. `make eat-psa-claim-limits-test` covers exact and plus-one configured claim limits. The verify-only, full-#tfm receiver is exercised by `make psa-eat-lean-verify`.

The pinned `t_cose` and QCBOR CI harness covers COSE wire interoperation in both directions for current-profile Sign1 and Mac0 payloads. `t_cose` validates the envelope produced by `wolfCOSE`; `wolfCOSE` validates a `t_cose` envelope and then performs the PSA/EAT claim checks. See `tests/interop/t_cose/README.md`.

9 MISRA C Compliance

wolfCOSE strives for MISRA C compliance and is checked in CI on every pull request. The project is not yet fully MISRA C:2023 compliant since there is no checker for 2023 that is publicly available yet. We do fully test and support the 2012 Misra-C standard. Full MISRA C:2023 verification requires commercial tooling (like LDRA, Polyspace, Cppcheck Premium). The current free tooling provides around 80% coverage across syntax, essential type, and data-flow rules.

9.1 Coverage

9.1.1 MISRA C:2012

Verified via cppcheck's MISRA addon (`--addon=misra`) with all wolfCOSE algorithm and feature macros defined. Covers ~65-75% of decidable MISRA C:2023 rules including:

- Syntax rules (comments, declarations, expressions)
- Control flow rules (goto, switch, loops)
- Declaration and type rules
- Pointer and array rules

Workflow: `.github/workflows/misra-2012.yml`

All wolfCOSE and wolfSSL feature macros are explicitly defined so cppcheck checks the full code path rather than enumerating wolfSSL's hundreds of platform `#ifdef` configurations. This includes all four default-off experimental COSE-HPKE operation gates, `WOLFCOSE_EXPERIMENTAL`, wolfSSL HPKE support, and `WOLFCOSE_ENABLE_DEPRECATED_ALGS`, so draft HPKE code and opt-in RFC 9053 algorithm paths are analyzed rather than silently excluded. See [Macros](#) for the complete list.

9.1.2 MISRA C:2023

MISRA C:2023 is essentially MISRA C:2012 plus Amendments 1-4. Since free tooling does not directly support MISRA C:2023, compliance is approximated using two approaches:

Strict Compiler Warnings 17 MISRA-adjacent GCC flags beyond the project's standard `-Wall -Wextra -Wpedantic -Wshadow -Wconversion`:

Flag	MISRA Rule	Purpose
<code>-Wcast-qual</code>	Rule 11.8	Const qualifier preservation
<code>-Wstrict-prototypes</code>	Rule 8.2	Function declaration completeness
<code>-Wmissing-prototypes</code>	Rule 8.2	Missing function prototypes
<code>-Wold-style-definition</code>	Rule 8.2	Old-style function definitions
<code>-Wdeclaration-after-statement</code>	Rule 8.x	C90-style declaration ordering
<code>-Wundef</code>	Rule 20.9	Undefined macro in <code>#if</code>
<code>-Wfloat-equal</code>	Rule 13.4	Float equality comparison
<code>-Wpointer-arith</code>	Rule 18.x	Pointer arithmetic safety
<code>-Wredundant-decls</code>	Rule 8.x	Redundant declarations
<code>-Wnested-externs</code>	Rule 8.x	Nested extern declarations
<code>-Wformat=2</code>	Rule 21.6	Format string safety
<code>-Wformat-security</code>	Rule 21.6	Format string security
<code>-Wlogical-op</code>	Rule 12.x	Logical vs bitwise operators
<code>-Wjump-misses-init</code>	Rule 15.1	Goto skipping initialization
<code>-Wdouble-promotion</code>	Rule 10.x	Implicit float-to-double
<code>-Wnull-dereference</code>	Safety	Null pointer dereference
<code>-Wsign-conversion</code>	Rule 10.x	Signed/unsigned conversion

clang-tidy Checks covers standard library safety (Amendment 3), deep data-flow analysis, and CERT C rules that map to MISRA 2023 logic:

Check Category	Coverage
bugprone-*	Unsafe patterns (buffer overflows, integer overflows, suspicious constructs)
cert-*	CERT C rules mapping to MISRA 2023 logic
clang-analyzer-*	Inter-procedural data-flow analysis, pointer tracking
misc-*	Miscellaneous safety checks (excluding misc-include-cleaner)

Workflow: `.github/workflows/misra-2023.yml`

9.2 Coverage Summary

Area	cppcheck (2012)	Compiler Flags + clang-tidy (2023)	Commercial Tools
Syntax Rules	High (~90%)	High (~95%)	100%
Essential Types	Medium (~50%)	High (~80%)	100%
Data Flow	Low (~30%)	Medium (~50%)	100%
Std Lib Safety	Low (~20%)	Medium (~60%)	100%

9.3 MISRA C:2012 Rule Enforcement

The MISRA C:2012 workflow does not suppress any project rule IDs. It retains the raw analyzer output, separately reports approved deviations and proven analyzer limitations, and fails every undeviated finding in `src/` or `include/wolfcose/`. The remaining cppcheck suppressions only exclude generic analyzer diagnostics and findings inside third-party wolfSSL headers.

The analyzer includes `tests/misra_consumer.c` as a downstream translation unit. This gives public APIs and public constants a real external use for Rules 8.7 and 2.5 without linking analyzer-only code into the library. For Rule 2.5, cppcheck does not count a macro used only by an `#if` or `#ifdef`; the report classifier accepts that limitation only after finding the exact macro in a tracked conditional preprocessing directive.

9.3.1 Approved deviations

ID	Rule	Location	Rationale and validation
D-11.5-001	11.5	<code>wolfCose_ForceZero</code>	Converting the caller's object pointer to a volatile character pointer is the standard C mechanism for securely erasing its object representation. Character access is alignment-safe and the function allocates no memory. <code>make zeroize-test</code> , <code>make zero-alloc-check</code> , and <code>make c99-check</code> validate the implementation.

ID	Rule	Location	Rationale and validation
D-19.2-001	19.2	WOLFCOSE_KEY.key	The public, discriminated union preserves the established ABI and embedded-memory footprint. <code>key</code> and <code>attachedType</code> govern member access. Replacing it with a structure would break ABI and increase RAM. The full tests exercise the supported members, and CI anchors the deviation to the exact union boundaries.
D-11.8-002	11.8	<code>wolfCose_Hpke0Seal</code> plaintext input	The pinned <code>wolfSSL wc_HpkeSealBase()</code> interface accepts a mutable plaintext pointer even though its implementation only consumes plaintext while writing to its distinct ciphertext output. The wrapper keeps the <code>wolfCOSE</code> public input <code>const</code> -qualified and avoids a payload-sized copy in constrained builds. The deviation is anchored to the one interoperability cast and covered by the HPKE unit and sanitizer CI lanes.
D-11.8-003	11.8	<code>wolfCose_Hpke0Open</code> ciphertext input	The pinned <code>wolfSSL wc_HpkeOpenBase()</code> interface accepts a mutable ciphertext pointer even though its implementation only consumes ciphertext while writing to its distinct plaintext output. The wrapper keeps the <code>wolfCOSE</code> public input <code>const</code> -qualified and avoids a ciphertext-sized copy in constrained builds. The deviation is anchored to the one interoperability cast and covered by the HPKE unit and sanitizer CI lanes.

`scripts/misra-deviations.json` identifies only these source locations. The classifier also pins a hash of the public union declaration. It fails if an expected diagnostic disappears, an anchor or approved declaration changes, or an additional finding of either rule appears, so a deviation cannot silently broaden or become stale.

9.4 MISRA C:2023 Deviations (clang-tidy)

The following clang-tidy checks are suppressed in the MISRA 2023 workflow. GCC strict MISRA warnings are fully clean (0 warnings).

9.4.1 bugprone-branch-clone: Identical Consecutive Switch Branches

Location: `src/wolfcose_alg.c` (algorithm dispatch switches)

Justification: Different COSE algorithms intentionally map to the same `wolfCrypt` value. For example, ES512 and EdDSA both use `WC_HASH_TYPE_SHA512`, and A128GCM/A192GCM share the same nonce length. The switch branches are not bugs; they represent distinct algorithm IDs with identical cryptographic parameters.

9.4.2 bugprone-easily-swappable-parameters: Adjacent Parameters of Similar Types

Location: Multiple public API functions

Justification: Advisory warning about adjacent function parameters of the same type (e.g., `size_t payloadLen`, `size_t detachedLen`). Fixing requires reordering or wrapping parameters, which would break the public API. The parameter ordering follows RFC 9052 structure conventions.

9.5 Examples and Test Code

examples/ and tests/ are runnable demonstration and test programs, not part of the shippable library, and are held to the same style rules where it keeps them useful as reference implementations. They are clean of the rules the library observes (no goto, fixed-length-coordinate checks, const-qualified literal payloads, braced statement bodies, unsigned size arithmetic, explicit precedence). The remaining deviations are inherent to runnable demos:

9.5.1 Rule 21.6: Standard I/O (and 17.7 on its return value)

Location: examples/, tests/.

Justification: Demos and test harnesses print human-readable status and PASS/FAIL to the console with printf/fprintf; the ignored return value (Rule 17.7) is part of the same console-output use. A real integration replaces this one call with a platform output routine. Library code under src/ uses no standard I/O.

9.5.2 Rule 15.5: Multiple return / single point of exit

Location: examples/, tests/.

Justification: Demonstration code uses early returns for linear, readable top-to-bottom flow. The library itself observes single-exit with cascading if (ret == 0) and a single return.

9.5.3 Rule 2.5: Unused macro definitions

Location: examples/, tests/.

Justification: Same false positive as the library: CI passes explicit -D feature flags so cppcheck checks one code path, which makes the guarded-away feature #defines look unused.

9.5.4 Rules 8.6 / 5.9 / 8.9: External/internal identifier definitions

Location: examples/.

Justification: Each demo is a standalone program with its own main and demo_* helpers. cppcheck reports these when scanning all demo translation units in a single pass; each program links independently, so there is no real multiple-definition.

9.6 Fully Compliant Rules (Notable)

Rule	Status	Notes
Rule 15.1 (no goto)	Compliant	All functions use cascading if (ret == WOLFCOSE_SUCCESS) with a single return ret
Rule 15.5 (single exit)	Compliant	All functions have exactly one return statement
Rule 17.2 (no recursion)	Compliant	CBOR decoder uses iterative traversal with a bounded stack (WOLFCOSE_CBOR_MAX_DEPTH)
Rule 17.7 (check returns)	Compliant	All return values checked or explicitly cast to (void)

Rule	Status	Notes
Rule 12.1 (explicit precedence)	Compliant	All &&/ operands and mixed arithmetic explicitly parenthesized

10 Footprint and Performance

This page reports build-profile footprint and end-to-end COSE_Sign1 throughput for ES256 and ML-DSA-44.

Size method. One fixed key and payload. Built `-ffunction-sections -fdata-sections` and linked `-Wl,--gc-sections`, so unreachable functions are dropped; this is the real flash cost, not the whole-archive size. Sizes are code + rodata. Two numbers are reported: **glue** is the wolfCOSE COSE + CBOR engine alone (independent of the crypto backend); **total** is glue plus the minimal wolfCrypt build pulled in. Desktop: x86_64 Intel i9-11950H, GCC 14.2. On-device: NUCLEO-H563ZI (STM32H563ZI, Cortex-M33 @ 250 MHz). June 2026.

10.1 Library size (glue)

The wolfCOSE COSE engine plus its built-in CBOR engine, for one ES256 COSE_Sign1.

Build	COSE engine	CBOR engine	Total (glue)
Sign + verify (WOLFCOSE_LEAN)	5.1 KB	1.7 KB	6.8 KB
Verify-only (WOLFCOSE_LEAN_VERIFY)	3.5 KB	1.6 KB	5.1 KB

wolfCOSE uses a tiny purpose-built CBOR engine (1.6–1.7 KB) rather than a general-purpose one, which is most of why the whole library stays small. Stripping the signing path (verify-only) shrinks every part.

10.2 Total footprint with wolfCrypt

The crypto backend dominates the binary. ES256 COSE_Sign1, wolfCrypt built minimal from source.

Build	Total + wolfCrypt
Sign + verify	34.6 KB
Verify-only (WOLFCOSE_LEAN_VERIFY)	26.2 KB

The verify-only profile is the common on-device case: a device verifies signed firmware or attestation while signing happens off-device on a server or HSM. It removes signing and the RNG entirely while keeping full RFC 9052 verification. wolfCrypt holds FIPS 140-3 certificate #4718, so this is also a direct path to a validated build.

10.3 Build-profile footprint

The lean profiles (see [Macros](#) -> Build Profiles). *Glue* is the wolfCOSE engine alone; *total* adds the minimal wolfCrypt backend.

Profile	Algorithm	wolfCOSE glue	Total + wolfCrypt
WOLFCOSE_LEAN	ES256 sign + verify	6.8 KB	34.6 KB
WOLFCOSE_LEAN_VERIFY	ES256 verify-only	5.1 KB	26.2 KB
WOLFCOSE_LEAN_MLDSA	ML-DSA-44 sign + verify	6.6 KB	35.8 KB
WOLFCOSE_LEAN_VERIFY_MLDSA	ML-DSA-44 verify-only	4.6 KB	20.8 KB

10.3.1 Post-quantum at the same cost (ML-DSA-44, FIPS 204)

ML-DSA-44 build	wolfCOSE	wolfCrypt	Total	ES256 equivalent
sign + verify (WOLFCOSE_LEAN_MLDSA)	6.6 KB	29.2 KB	35.8 KB	34.6 KB
verify-only (WOLFCOSE_LEAN_VERIFY_MLDSA)	4.6 KB	16.2 KB	20.8 KB	26.2 KB

Post-quantum sign + verify lands within ~1 KB of classical ES256, and verify-only is actually *smaller* (20.8 vs 26.2 KB): wolfCOSE adds just 4.6 KB on top of wolfCrypt for ML-DSA verify, less than its ES256 glue, because ML-DSA skips the DER signature conversion ECDSA needs.

10.4 Speed (Intel i9-11950H, x86_64)

ES256 and ML-DSA-44 rates are measured end-to-end for COSE_Sign1 on an Intel i9-11950H using optimized wolfCrypt assembly. See the [published measurements](#) for more background.

Operation	Scope	Ops/s
ES256 verify	End-to-end COSE_Sign1	26,437
ES256 sign	End-to-end COSE_Sign1	66,538
ML-DSA-44 verify	End-to-end COSE_Sign1	53,686
ML-DSA-44 sign	End-to-end COSE_Sign1	21,986

10.5 On a real MCU: STM32H563 (Cortex-M33 @ 250 MHz)

ES256 COSE_Sign1 verify on a NUCLEO-H563ZI, verify-only with no RNG, timed with the on-chip DWT cycle counter. wolfCrypt uses its Cortex-M Thumb P-256 assembly (WOLFSSL_SP_ARM_CORTEX_M_ASM).

Metric	wolfCOSE + wolfCrypt
verify cycles/op	1,510,716
verify ops/s @ 250 MHz	~165
flash (verify-only)	44.2 KB
peak stack	6.7 KB
peak heap	0 (zero-alloc)
total RAM	~6.7 KB

The verify path allocates nothing, using only caller-provided buffers, so the entire RAM cost is ~6.7 KB of stack, with no heap at all, which matters where malloc is prohibited. The footprint is ~2% of the board's 2 MB flash and ~1% of its 640 KB RAM.

10.6 Method and versions

Desktop: x86_64 Intel i9-11950H, GCC 14.2. ES256 was measured in June 2026 using wolfCOSE 8c6209e and wolfSSL 4c0c093. ML-DSA-44 was measured in September 2026 using wolfCOSE 82f42a4 and wolfSSL 070d311. On-device: NUCLEO-H563ZI (STM32H563ZI, Cortex-M33 @ 250 MHz), arm-none-eabi-gcc 13.2, DWT cycle counter.

10.7 See Also

- **Macros:** build profiles and configuration
- **Testing:** memory and stack-bound enforcement in CI
- **Getting Started:** building wolfCOSE

11 Testing

wolfCOSE includes comprehensive testing infrastructure for unit tests, algorithm coverage, code coverage, and failure injection testing. CI requires 100% line coverage for every wolfCOSE source file. These thresholds are validated on every push and PR to ensure coverage does not regress.

11.1 Running Tests

11.1.1 Release Qualification

Release candidates add the `ci:release` label to run the read-only Release Qualification workflow. The workflow validates version metadata, exercises the optional feature profiles, audits every required CI workflow associated with the pull request head, builds reproducible source archives, and smoke-tests both archives. A manual post-merge run qualifies the exact commit used to build the release artifacts. Like wolfProvider's label-selected jobs, the workflow starts only when the label is added; remove and re-add the label to qualify a newer commit on the same pull request. Apply the label only after the candidate version and release notes are committed.

The release-only local targets are:

```
make release-scenarios
make release-coverage
make cxx-check
make valgrind-check
make release-validate VERSION=X.Y.Z RELEASE_REF=HEAD
make release-artifacts VERSION=X.Y.Z RELEASE_REF=HEAD
```

Replace X.Y.Z with the version in `include/wolfcose/version.h`.

`release-scenarios` combines HPKE one-recipient and multi-recipient delivery, stateful HSS/LMS signing, lean LMS verification, ML-DSA signing, RFC 9783 device onboarding, and delegated PSA/EAT signing. The HPKE scenario rejects modified authentication data, modified ciphertext, and a mismatched recipient. The LMS scenario signs two messages, proves the persisted state advances, and rejects a modified signature. The PSA/EAT scenario rejects a wrong challenge, a modified token, and an unapproved software measurement.

11.1.2 Basic Unit Tests

```
make test
```

This runs the full test suite including: - CBOR encoding/decoding tests (RFC 8949 Appendix A vectors) - COSE Sign1/Encrypt0/Mac0 tests - COSE Sign/Encrypt/Mac multi-party tests - Interoperability tests with COSE Working Group vectors

11.1.3 PSA/EAT Attestation Tests

```
make eat-psa-test
make eat-psa-float-test
make eat-psa-min-buffers-test
make eat-psa-claim-limits-test
make eat-psa-profile-test
make eat-psa-ext-sign-test
make eat-psa-config-check
make psa-eat-lean-verify
make psa-eat-demo
```

`eat-psa-test` builds the explicit full RFC 9783 profile and runs the Appendix A Sign1 and Mac0 vectors, current and legacy profile cases, current algorithm round trips, and malformed-token negatives. The vectors were generated by TF-M's `iat-verifier` as documented by RFC 9783.

`eat-psa-ext-sign-test` reruns the full profile with `WOLFCOSE_ENABLE_EXT_SIGN` to exercise the delegated (external) signing path for PSA/EAT tokens.

`eat-psa-min-buffers-test` reruns that full profile with `WOLFCOSE_MIN_BUFFERS`. It proves that the PSA/EAT claim parser accepts the complete current-profile fixture even when the ordinary COSE header-map limit is reduced for constrained targets.

`eat-psa-float-test` recompiles the full profile with `WOLFCOSE_FLOAT` and executes both floating-point encoder safety checks against a tolerant decoder context.

`eat-psa-claim-limits-test` recompiles a focused full profile with the minimum valid `WOLFCOSE_EAT_PSA_MAX_CLAIMS` and `WOLFCOSE_EAT_PSA_MAX_COMPONENT_CLAIMS` values. It accepts exact-size maps and rejects a top-level or component-map declaration one entry over its limit.

`eat-psa-profile-test` runs separate current Sign1-only, current Mac0-only, legacy Sign1-only, and legacy Mac0-only binaries. It proves partial current attesters can issue and generically authenticate #t fm with their enabled envelope, while their incomplete PSA/EAT receivers reject that profile. It also proves disabled standardized claim namespaces are rejected instead of skipped, and a structurally valid token using its compiled-out envelope returns `WOLFCOSE_E_UNSUPPORTED` before cryptographic verification.

`eat-psa-config-check` proves that the default library has no PSA/EAT symbols, then compiles minimal current and legacy Sign1/Mac0, issuance-only, and invalid gate combinations. It also checks the 384, 521, and above-P-521 `ECC_MIN_KEY_SZ` boundaries and rejects PSA/EAT resource limits below their structural floors. `psa-eat-lean-verify` runs the RFC Sign1 and Mac0 vectors against a verify-only, full-#t fm receiver build. `psa-eat-demo` runs a complete issue, challenge verification, and software-component appraisal workflow. See [PSA-EAT](#).

11.1.4 Deprecated Algorithm ID Tests

`make deprecated-algs-test`

Rebuilds the suite with `WOLFCOSE_ENABLE_DEPRECATED_ALGS` so the RFC 9053 ES256/ES384/ES512/EdDSA paths, the COSE WG example vectors, and the RFC 9783 PSA tokens run. CI runs it as a dedicated step, and the minimal-build matrix also runs the full suite with the macro enabled.

11.1.5 CLI Tool Tests

`make tool-test`

Round-trip self-tests for all 17 supported CLI algorithms. Each algorithm is tested with key generation, operation, and verification.

11.1.6 Experimental COSE-HPKE Tests

COSE-HPKE P0 is off by default and therefore has a dedicated opt-in test path. Build against a wolfSSL configured with `--enable-hpke --enable-ecc --enable-aesgcm --enable-keygen`, then acknowledge experimental draft support and enable the four operation macros:

```
HPKE_FLAGS="-DWOLFCOSE_EXPERIMENTAL \
-DWOLFCOSE_ENABLE_HPKE_0_ENCRYPT \
-DWOLFCOSE_ENABLE_HPKE_0_DECRYPT \
-DWOLFCOSE_ENABLE_HPKE_0_KEY_ENCRYPT \
-DWOLFCOSE_ENABLE_HPKE_0_KEY_DECRYPT"
```

```
make test EXTRA_CFLAGS="$HPKE_FLAGS"
make hpke-demo EXTRA_CFLAGS="$HPKE_FLAGS"
make c99-hpke-check WOLFSSL_INC=/path/to/hpke-enabled-wolfssl/include
EXPECT_HPKE=true make cmdline-test EXTRA_CFLAGS="$HPKE_FLAGS"
```

The command-line test performs public/private key export, rejects identical, normalized-alias, case-alias (where the filesystem supports it), and symlink-alias key destinations, preserves existing POSIX destinations, and rejects a failed paired key export without leaving a private-key file. It rejects a plaintext one byte above the configured maximum for both constructions, exercises HPKE-0 integrated encryption, two-recipient HPKE-0-KE encryption and decryption at both recipient indices, maximum-message round trips for both constructions, and a focused test - a round trip for each construction. The dedicated strict C99 target compiles every HPKE-gated library, test, tool, and example path under each one-way operation gate, both convenience gates, the complete P0 configuration, and wolfSSL's NO_ECC256 plus HAVE_ALL_CURVES configuration. It also rejects HPKE when NO_ECC_SECP or an ECC_MIN_KEY_SZ above 256 makes P-256 unavailable, or when NO_AES_128 removes the fixed suite's AEAD key size. Every enabled configuration also supplies WOLFCOSE_EXPERIMENTAL; the regular make experimental-check target verifies that an HPKE operation selected without that acknowledgement is rejected at compile time. The unit suite validates draft messages that omit the optional HPKE alg (including a second selected recipient), rejects an HPKE alg in an unprotected header, accepts an unprotected HPKE-0-KE content algorithm only when its authenticated Recipient_structure binds it, and rejects a modified content algorithm. The CI backend enables Koblitz curves to prove that a 32-byte secp256k1 key cannot masquerade as P-256 at the HPKE API boundary. It also covers missing, duplicate, wrong-type, wrong-length, and wrongly placed ek, prohibited psk_id, detached ciphertext, and cleared outputs on failed decrypts and encrypts. GitHub Actions runs the same coverage in [Experimental COSE-HPKE](#).

11.1.7 Comprehensive Algorithm Tests

```
make comprehensive
```

Runs ~240 algorithm combination tests covering: - All signature algorithms with various payloads - All encryption algorithms with various key sizes - All MAC algorithms - Multi-signer and multi-recipient combinations - Error handling and edge cases

11.1.8 Scenario Examples

```
make scenarios
```

Runs real-world scenario examples: - Firmware signing with ML-DSA - Multi-party approval workflows - IoT fleet configuration - Sensor attestation - Group broadcast MAC

11.1.9 Interoperability (t_cose)

```
make interop-tcose \
  TCOSE_DIR=/path/to/t_cose QCBOR_DIR=/path/to/QCBOR \
  TCOSE_CRYPTOLIB="-lcrypto"
```

Proves RFC 9052 wire interoperability between wolfCOSE (on wolfCrypt) and t_cose (on OpenSSL): each library produces COSE messages the other consumes, both directions, across every algorithm both implement — ES256/384/512, PS256/384/512, EdDSA (Ed25519, Ed448), HMAC 256/384/512, and AES-GCM 128/192/256. The bytes on the wire are the only interface; the two APIs are never reconciled. Each primitive class also exercises a tamper case that wolfCOSE must reject.

The same pinned harness also performs current-profile PSA/EAT Sign1 and Mac0 interoperability. t_cose verifies wolfCOSE's envelope bytes; wolfCOSE verifies the t_cose envelope and validates the

RFC 9783 claims.

t_cose and QCBOR are BSD-3-Clause and are not vendored; the [Interop CI job](#) fetches them at pinned SHAs. See `tests/interop/t_cose/README.md` for the fixed test-key provenance.

11.1.10 Complete upstream t_cose suite

```
make tcose-upstream \
  TCOSE_DIR=/path/to/t_cose QCBOR_DIR=/path/to/QCBOR
```

This runs the entire pinned upstream t_cose suite, including its API-level and QCBOR-adapter tests. At the current pinned revision it runs 43 tests. It is a dependency-health gate, not wolfCOSE API coverage: t_cose's internal API tests cannot be redirected to wolfCOSE. Use `interop-tcose` for wolfCOSE wire-format compatibility.

11.1.11 Interoperability (go-cose)

```
make interop-go-cose
```

This runs live, bidirectional COSE_Sign1 interop against [Veraison go-cose](#), pinned at v1.3.0 in `tests/interop/go_cose/go.mod`. The matrix covers ES256, ES384, ES512, PS256, PS384, PS512, Ed25519, ES256 with external AAD, and untagged ES256. The pinned peers predate RFC 9864, so every interop target links a wolfCOSE built with `WOLFCOSE_ENABLE_DEPRECATED_ALGS` and still exchanges the RFC 9053 IDs. Each implementation signs a message the other verifies, validates the payload, and rejects a modified signature. Go 1.21 or later is required. go-cose requires an embedded payload for Sign1 verification, so detached Sign1 remains covered by the wolfCOSE tests and `interop-tcose`. go-cose's scope makes it a Sign and Sign1 oracle; `interop-tcose` remains the broader Mac0 and Encrypt0 wire suite. The target also verifies RFC 9783's ES256 PSA token and decodes its standard EAT claims with Go CBOR.

11.1.12 Interoperability (python-cwt)

```
python3 -m pip install -r tests/interop/python_cwt/requirements.txt
make interop-python-cwt
```

This runs live recipient-message interop against [python-cwt](#), pinned at 3.3.0 with all Python dependencies locked in `requirements.txt`. It exchanges A128GCM direct COSE_Encrypt, ECDH-ES plus HKDF-SHA-256 COSE_Encrypt, and HMAC-256 direct COSE_Mac messages in both directions. Each message carries external AAD, checks its payload, and must reject a modified authenticated byte. The target also verifies RFC 9783's ES256 PSA token and decodes selected EAT claims with `python-cwt` and `cbor2`.

A128KW COSE_Encrypt interoperates in both directions. wolfCOSE emits an empty protected recipient bucket with the algorithm in the unprotected header, per RFC 9053 Section 6.2.1, which `python-cwt` 3.3.0 both produces and accepts.

11.1.13 Interoperability (Rust coset)

```
make interop-rust-coset
```

This runs live, bidirectional COSE_Sign1 interop against [Google coset](#), pinned at 0.4.2 in `tests/interop/rust_coset/Cargo.lock`, using RustCrypto for signature operations. It covers ES256, Ed25519, ES256 with external AAD, untagged ES256, and detached ES256. Both peers validate the protected algorithm and payload semantics, then reject a modified signature. Rust 1.81 or later is required.

11.1.14 COSE WG Examples vectors

`make test` includes a curated, fixed subset from the [COSE WG Examples repository](#), pinned at commit 53c9d634333bb4f529d78f5980fffa2667ee2c12. It verifies ES256 COSE_Sign1 and COSE_Sign, HMAC-256 COSE_Mac0, A128GCM COSE_Encrypt0, and multi-recipient A128GCM direct, A128KW, and ECDH-ES COSE_Encrypt vectors. Every selected vector checks its cleartext and rejects a modified authentication value. The selected Mac0 vectors carry alg only in an unprotected header, so the test pins the expected algorithm on the local key instead of accepting an unconstrained algorithm from the message.

11.1.15 PSA attestation-token acceptance

`make test` includes fixed acceptance vectors from RFC 9783 Appendix A. The tests verify ES256 COSE_Sign1 and HMAC-256 COSE_Mac0 PSA attestation tokens, reject a modified authentication value, and decode the EAT/PSA claim payload with wolfCOSE CBOR APIs.

11.2 Code Coverage

11.2.1 Running Coverage

`make coverage`

`make eat-psa-coverage`

`coverage` keeps the default PSA/EAT-free library configuration. The explicit `eat-psa-coverage` target enables the full RFC 9783 receiver profile before collecting the PSA/EAT source coverage report.

11.2.2 Coverage Targets

Component	Target
Every <code>src/*.c</code> file	100%

11.2.3 Coverage with Failure Injection

`make coverage-force-failure`

`make eat-psa-coverage-force-failure`

The `eat-psa-*` form runs the corresponding full PSA/EAT profile. Both forms enable additional coverage by testing error paths that normally require wolfCrypt internal failures.

11.3 Force Failure Testing

wolfCOSE includes a failure injection system for testing error paths that are difficult to reach through normal testing.

The `WOLFCOSE_FORCE_FAILURE` build flag enables controlled injection of failures at specific points in the code. This allows testing of:

- Crypto operation failures (signature, encryption, decryption, MAC)
- Key operation failures
- Memory/buffer errors
- Internal state errors

11.3.1 Production Builds

The force failure system compiles out completely in production builds. When `WOLFCOSE_FORCE_FAILURE` is not defined:

- All failure injection code is excluded
- `wolfForceFailure_Check()` always returns 0
- No runtime overhead

11.4 CI Pipeline

wolfCOSE runs the following CI checks on every push and pull request:

11.4.1 Build and Test Matrix

Environment	Compilers
Ubuntu (latest + 22.04)	GCC 10, 11, 12, 13, 14
Ubuntu (latest + 22.04)	Clang 14, 15, 16, 17, 18
macOS	Xcode default

11.4.2 Test Stages

1. **Build:** Compile library and tests
2. **Unit Tests:** Run CBOR and COSE test suites
3. **Comprehensive Tests:** ~240 algorithm combination tests
4. **Scenario Examples:** Real-world workflow tests
5. **Tool Tests:** CLI round-trip tests (17 algorithms)
6. **Experimental COSE-HPKE:** Opt-in P0 unit tests, example, CLI commands, and both construction-specific self-tests

The Interop workflow intentionally keeps `t_cose`, `go-cose`, `python-cwt`, Rust `coset`, and the complete upstream `t_cose` suite in one matrix job per wolfSSL version. They share the same wolfSSL build; separate jobs would only duplicate that setup without providing useful parallelism.

11.4.3 Memory and Stack Bounds

wolfCOSE is zero-heap (no `malloc/XMALLOC` on any path) and bounded-stack, both enforced in CI:

- **Bounded stack:** built with `-fstack-usage`, then `scripts/check_stack_usage.sh` fails the build if any wolfCOSE frame exceeds 6144 bytes or is dynamic (unbounded); `-Werror=vla` bans VLAs/`alloca`.
- **Zero heap:** sources, tests, tools, and examples are grepped for allocator calls.
- **WOLFCOSE_MIN_BUFFERS:** constrained-target profile that shrinks the caller working buffers (not the library frames) — see [Macros](#).
- **Minimal Build matrix:** builds and tests against single-purpose minimal wolfCrypt configs (ECC-only, EdDSA-only, AEAD-only, MAC-only, ...) plus a `WOLFCOSE_LEAN` core build.

11.4.4 Lean and Post-Quantum Builds

The Lean Build workflow (`.github/workflows/lean-build.yml`) exercises the minimal on-device build profiles:

Job	What it checks
Lean verify-only	Builds + runs examples/sign1_verify_lean.c with WOLFCOSE_LEAN_VERIFY against a minimal ECC-only wolfSSL; asserts the signing API is not linked.
Lean configs compile clean	Strict -Werror compile of the full, WOLFCOSE_LEAN_VERIFY, sign-only, WOLFCOSE_LEAN_MLDSA, and WOLFCOSE_LEAN_VERIFY_MLDSA configurations.
Post-quantum ML-DSA	Builds wolfSSL with ML-DSA and runs make mldsa-demo (sign+verify) and make mldsa-verify (verify-only); asserts the verify-only build links no signing API.

11.4.5 Static Analysis

Tool	Purpose
cppcheck	Static code analysis
Clang Static Analyzer	Data flow analysis
GCC -fanalyzer	GCC's built-in analyzer
Advanced Internal Static Analysis	Security Audit
In PR Opus 4.6 Diff review with wolfSSL internal review bot	Security Audit

11.4.6 Coverity Scan

Nightly defect analysis via [Coverity Scan](#).

11.5 Test File Structure

```
tests/
  test_cbor.c      # CBOR vectors (RFC 8949 Appendix A) + round-trip
  test_cose.c      # COSE Sign1/Encrypt0/Mac0/Sign/Encrypt/Mac tests
  test_interop.c   # Interoperability tests with RFC vectors
  test_main.c      # Test harness (CI exit codes)
  force_failure.c  # Failure injection implementation
  force_failure.h  # Failure injection API
  vectors/         # Test vectors from COSE Working Group
```

11.5.1 Test Categories in test_cose.c

Category	Description
Sign1 Tests	Single-signer signature creation and verification
Encrypt0 Tests	Symmetric encryption and decryption
Mac0 Tests	Symmetric MAC creation and verification
Sign Tests	Multi-signer messages
Encrypt Tests	Multi-recipient encryption
Mac Tests	Multi-recipient MAC
Key Tests	COSE_Key encoding and decoding
Error Tests	Invalid inputs, tampered messages
Detached Payload Tests	Messages with external payloads

Category	Description
External AAD Tests	Additional authenticated data

11.6 Test Vectors

The tests/vectors/ directory contains test vectors from: - COSE Working Group examples - RFC 9052 examples - Custom edge case vectors

Vector format is typically CBOR diagnostic notation or hex dumps with expected outputs.

11.7 See Also

- [Getting Started](#): Build instructions
- [Macros](#): Test configuration macros
- [Project Structure](#): Source file layout

12 STM32Cube

wolfCOSE ships as an STM32Cube pack, I-CUBE-wolfCOSE, so it drops into any STM32CubeMX or STM32CubeIDE project without manual source integration. The pack provides wolfCOSE as an STM32 middleware and uses the wolfSSL pack (I-CUBE-wolfSSL) for all cryptography.

Supported cores: Cortex-M0, M0+, M3, M4, M7, M23, M33, M55, and STM32MP1.

12.1 Prerequisites

- STM32CubeMX, plus STM32CubeIDE or another toolchain to build.
- The wolfSSL pack, I-CUBE-wolfSSL 5.9.2 or later. wolfCOSE depends on wolfCrypt for hashing, signing, and AEAD.
- A board with a hardware RNG and a UART for console output.

12.2 Install the packs

1. Download the wolfSSL pack ([I-CUBE-wolfSSL.pack](#)) and the wolfCOSE pack ([I-CUBE-wolfCOSE.pack](#)).
2. In STM32CubeMX, open Help, Manage embedded software packages, From Local..., and install the wolfSSL pack first, then the wolfCOSE pack.

12.3 Add wolfCOSE to a project

1. Open or create a project .ioc for your board.
2. Enable the RNG peripheral under Pinout & Configuration, Security, RNG. Signing needs entropy, and wc_GenerateSeed() fails without it.
3. Open Software Packs, Select Components.
4. Enable wolfSSL wolfCrypt Core and wolfCOSE Core. To run the on device self test, also enable wolfCOSE Test.
5. In the Software Packs configuration category, enable each pack.
6. Generate code and build with your toolchain.

12.4 Configure algorithms

wolfCOSE reads its configuration from the wolfSSL user_settings.h, included before wolf-cose/settings.h. Enable the wolfCrypt features that match the COSE algorithms you use:

- ES256: HAVE_ECC, WOLFSSL_SHA256
- ES384 or ES512: add WOLFSSL_SHA384 or WOLFSSL_SHA512
- ML-DSA (RFC 9964): WOLFSSL_HAVE_MLDSA
- Encrypt0 AEAD: HAVE_AESGCM, or HAVE_CHACHA with HAVE_POLY1305
- MAC0: HMAC, which is on by default with SHA support

If a required wolfCrypt feature is missing, wolfcose/settings.h raises a compile error naming it.

12.5 Run on a device

The Test component builds wolfcose_test.c, a self test that runs a COSE_Sign1 sign and verify and reports the result over your configured console. Call wolfCOSETest() from your application once main has initialized the clocks and console.

A ready to run example for the NUCLEO-H563ZI board lives in [wolfssl-examples-stm32](#), with a pre-configured .ioc: install the packs, open the .ioc, generate, add the glue and software-crypto config from that example's README, build, flash, and watch the console.

Verified on NUCLEO-H563ZI hardware, the console prints:

```
== wolfCOSE NUCLEO-H563ZI ==  
Running wolfCOSE test (COSE_Sign1 ESP256)...  
wolfCOSE test: PASS (COSE_Sign1 99 bytes)
```

12.6 Notes

- Only pack source is kept in the repository. The built pack is posted at [wolfssl.com/files/ide/I-CUBE-wolfCOSE.pack](https://www.wolfssl.com/files/ide/I-CUBE-wolfCOSE.pack).
- On some STM32 families the wolfSSL pack enables hardware hash and RNG by default. If a build reports a missing HAL module or hash symbol, enable the matching peripheral in the `.ioc`, or select software crypto with `NO_STM32_HASH` and `NO_STM32_RNG` in the generated `wolfSSL.I-CUBE-wolfSSL_conf.h`. On the STM32H5 the hardware hash block references a HAL enum that does not exist, so software crypto is required there; the [NUCLEO-H563ZI example](#) shows the exact edit.

13 Experimental Features

This page tracks wolfCOSE features that intentionally remain outside the stable default build while their standards are still in development.

Status snapshot: 2 September 2026. Update this page when the IETF draft, its IANA registrations, or wolfCOSE's implementation status changes.

13.1 Customer Summary

Experimental code is disabled by default. Enabling a feature requires both the master acknowledgement, `WOLFCOSE_EXPERIMENTAL`, and the feature's individual `WOLFCOSE_ENABLE_*` macro. This makes draft adoption an explicit build-time decision and keeps ordinary deployments free of draft wire formats and APIs.

Experimental features may change or be removed before their associated RFC is published. Customers who evaluate one should pin the wolfCOSE revision used for testing and should not promise long-term wire compatibility until the feature graduates.

13.2 COSE-HPKE

13.2.1 Standards Status

Item	Current status
Specification	draft-ietf-cose-hpke-26
IETF state	Active COSE working-group Internet-Draft
Intended RFC status	Proposed Standard
Published	4 July 2026
Draft expiration	5 January 2027
Stable RFC	Not published

Draft expiration is not an RFC publication estimate. The IETF may revise, replace, or advance the document at any time. Track the [current Datatracker record](#) for the authoritative state.

13.2.2 wolfCOSE Status

Area	Status	Scope
Master acknowledgement	Complete	<code>WOLFCOSE_EXPERIMENTAL</code> is required but enables no code by itself.
Fine-grained build controls	Complete	Separate send and receive macros exist for integrated HPKE-0 and multi-recipient HPKE-0-KE.
Integrated encryption	Complete for P0	<code>COSE_Encrypt0</code> HPKE-0 encrypt and decrypt.
Multi-recipient encryption	Complete for P0	<code>COSE_Encrypt</code> HPKE-0-KE encrypt and decrypt.

Area	Status	Scope
Command-line and example coverage	Complete for P0	HPKE commands, self-tests, and examples/hpke_demo.c use the same experimental gate.
Continuous integration	Complete for P0	Unit, example, CLI, strict C99, malformed-input, and curve-validation coverage run in the experimental HPKE workflow.
Stable wire and API commitment	Pending	Deferred until the final RFC and IANA values are published.

13.2.3 Implemented P0 Scope

The current implementation is deliberately narrow and auditable:

- HPKE base mode only.
- DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, and AES-128-GCM only.
- One-recipient integrated COSE_Encrypt0 HPKE-0.
- Multi-recipient COSE_Encrypt HPKE-0-KE with an independently protected CEK for each recipient.
- External AAD bound through the normal COSE encryption structure.

The following are not implemented in P0: alternate KEM, KDF, or AEAD suites; X25519; PSK mode; authenticated HPKE mode; and an application-controlled HPKE info value. These omissions are intentional, not hidden runtime options.

13.2.4 Build Selection

Build wolfSSL with HPKE, ECC P-256, AES-128-GCM, SHA-256, and key generation, then select only the wolfCOSE directions the product needs:

```
# Integrated Encrypt0, send and receive.
make EXTRA_CFLAGS="-DWOLFCOSE_EXPERIMENTAL -DWOLFCOSE_ENABLE_HPKE_0"

# Receive-only integrated provisioning target.
make EXTRA_CFLAGS="-DWOLFCOSE_EXPERIMENTAL \
-DWOLFCOSE_ENABLE_HPKE_0_DECRYPT"

# Multi-recipient sender in a lean build.
make EXTRA_CFLAGS="-DWOLFCOSE_LEAN -DWOLFCOSE_EXPERIMENTAL \
-DWOLFCOSE_ENABLE_ENCRYPT -DWOLFCOSE_ENABLE_HPKE_0_KE_ENCRYPT"
```

The make hpke-demo target supplies the acknowledgement and all four P0 operation macros itself. See [Configuration Macros](#) for every enable and compile-out macro.

13.2.5 Path to Stable Support

Milestone	Status	Customer planning guidance
New draft revisions	Ongoing	wolfCOSE will assess each revision for wire, identifier, and API changes.

Milestone	Status	Customer planning guidance
Final RFC and IANA registrations	Pending IETF publication	No public RFC publication date is available. Do not use the draft expiration as a delivery date.
wolfCOSE compatibility update	Pending final RFC	Planned after the final RFC and registrations are available.
Graduation from experimental	Pending validation	The target is a compatible wolfCOSE release after final-RFC validation. The release date cannot be committed before the IETF publication date is known.

When COSE-HPKE graduates, wolfCOSE will publish migration notes, update the implemented identifiers and wire handling as needed, remove the WOLFCOSE_EXPERIMENTAL requirement, and document the stable API and compatibility policy. Until then, use a COSE signature or MAC with HPKE when the sender must be authenticated, and treat HPKE-encoded artifacts as experimental interoperability data.

14 Release Notes

14.1 wolfCOSE 2.0.0 (September 17, 2026)

Release 2.0.0 adds standardized HSS/LMS signatures, COSE countersignatures, PSA/EAT attestation, RFC 9864 fully specified signature algorithms, experimental COSE-HPKE, delegated signing, expanded key and CBOR APIs, and extensive parser and key-handling hardening. wolfCOSE remains a zero-allocation C implementation of CBOR and COSE backed solely by wolfCrypt.

This is a major release because WOLFCOSE_KEY gained fields since 1.0.0. Applications must be recompiled against the 2.0.0 headers. Code that initializes keys with `wc_CoseKey_Init()` and attaches keys through the `wc_CoseKey_Set*()` APIs does not need source changes.

14.1.1 Vulnerabilities

- No CVEs were assigned for this release.

14.1.2 New Feature Additions

- Added delegated signing for COSE_Sign1 and multi-signer COSE_Sign. WOLFCOSE_ENABLE_EXT_SIGN lets an HSM, secure element, PSA Crypto service, or remote KMS produce signatures without exposing the private key to wolfCOSE. (#59)
- Added `wc_CoseSign1_Sign_ex()` for tagged or untagged output and `wc_CoseSign1_SignSize_ex()` for exact sizing without signing, consuming an RNG, or advancing an external signer. (#65)
- Expanded the COSE_Key API with public-only encoding, exact encoded-size queries, raw ECC coordinate encoding, and metadata inspection before key import. Added CBOR context initializers, int-or-text label decoding, and zero-copy skipped-item capture helpers. (#66)
- Added RFC 8778 HSS/LMS signatures and COSE_Key support for COSE_Sign1, multi-signer COSE_Sign, delegated signing, and lean sign/verify profiles. Signing state remains owned and persisted by the caller. (#72)
- Added RFC 9338 full and abbreviated countersignatures across all six tagged COSE message types, including in-place creation, independent verification, CLI support, and RFC interoperability vectors. (#74)
- Added RFC 9783 PSA/EAT token issuance, verification, UEID-selected key resolution, software-component iteration, delegated signing, current and legacy profile handling, and Sign1 and Mac0 envelopes. PSA/EAT remains explicitly enabled by application-selected compile-time gates. (#75)
- Added RFC 9864 fully specified signature algorithms ESP256, ESP384, ESP512, Ed25519, and Ed448 as the default IDs. The deprecated RFC 9053 polymorphic IDs remain available with WOLF-COSE_ENABLE_DEPRECATED_ALGS. (#85)
- Added a two-step compile-time gate for draft features. A draft feature must select its own operation gate and acknowledge unstable specifications with WOLFCOSE_EXPERIMENTAL. (#68)
- Added experimental COSE-HPKE P0 support for COSE_Encrypt0 and multi-recipient COSE_Encrypt, including CLI commands and tests. It is off by default and requires WOLFCOSE_EXPERIMENTAL plus the selected HPKE operation gates while the specification remains an Internet-Draft. (#70)
- Added the I-CUBE-wolfCOSE STM32Cube pack, STM32CubeMX/IDE integration documentation, and an on-device test. (#61)

14.1.3 Fixes

- Rejected COSE_Key decoding into an attached wolfCrypt key object of a different type before any import occurs. Thanks to Omoikane Labs for the report. (#64)
- Hardened CBOR and COSE parsing, cryptographic dispatch, length handling, strict serialization checks, key encoding and decoding, and cleanup on error paths. (#67)

- Rejected invalid protected and skipped recipient structures, including nested Direct recipients and conflicting recipient metadata. (#77)
- Tightened COSE_Key metadata validation, public/private material checks, constant-time ML-DSA public-key comparison, and attached-key type and curve validation across signing, verification, and ECDH operations. (#78, #80)
- Hardened detached input checks, critical-header handling, ECDSA signing prerequisites, protected-header pointers, AES cleanup, CBOR state tracking, public API contracts, and recipient depth accounting. (#81, #82)
- Hardened RSA COSE_Key import and export by validating component presence and widths, bounding output by RSA capacity, fully initializing modulus buffers, and requiring hardened private-key import behavior. (#83)
- Guarded COSE header label conversion against values outside the supported signed range. (#86)

14.1.4 Improvements/Optimizations

- Split the former monolithic implementation into per-area source modules for smaller selective builds and easier analysis. (#73)
- Added pkg-config discovery while retaining explicit wolfSSL include and library overrides. (#69)
- Expanded interoperability testing against t_cose/QCBOR, go-cose, python-cwt, Rust coset, OpenSSL, COSE WG examples, and RFC vectors. (#71)
- Tightened internal integer and size types for MISRA C analysis and fixed static-analysis stability in the test suite. (#62, #79)
- Added optional RFC 6979 deterministic ECDSA signing through WOLFCOSE_ENABLE_DETERMINISTIC_ECDSA. (#67)
- Refreshed the README, algorithm documentation, build guidance, and measured ES256 and ML-DSA-44 performance and footprint references. (#87, #88)
- Added release-only qualification targets for advanced feature scenarios, C++ compilation, Valgrind, merged coverage profiles, metadata validation, and reproducible smoke-tested source archives. (#89)

14.1.5 Compatibility / Migration

- WOLFCOSE_KEY gained delegated-signing and internal key-type tracking fields. Recompile applications against the 2.0.0 headers.
- New messages use the RFC 9864 fully specified signature IDs. Existing messages using ES256, ES384, ES512, or EdDSA require WOLFCOSE_ENABLE_DEPRECATED_ALGS; changing a protected algorithm ID also requires re-signing the message. See [Migrating from RFC 9053 Signature IDs](#).
- COSE-HPKE is experimental. Its wire format and API may change until the IETF specification is finalized.
- LIBWOLFCOSE_VERSION_STRING is now "2.0.0" and LIBWOLFCOSE_VERSION_HEX is 0x02000000.

14.2 wolfCOSE 1.0.0 (June 25, 2026)

Release 1.0.0 is the first stable release of wolfCOSE, a complete, zero-allocation C implementation of CBOR (RFC 8949) and COSE (RFC 9052/9053) on top of wolfCrypt. It provides all six COSE message types in both single-actor and multi-actor forms, 40 algorithms across signing, encryption, MAC, and key distribution, and standardized post-quantum ML-DSA signatures (RFC 9964), all heap-allocation-free and within a tiny footprint.

14.2.1 Vulnerabilities

- None. This is the initial release.

14.2.2 New Feature Additions

- CBOR engine implementing RFC 8949 encode/decode with no external dependency, enforcing deterministic/preferred-encoding rules and rejecting non-preferred or trailing input on decode.
- All six COSE message types (RFC 9052): COSE_Sign1, COSE_Sign, COSE_Encrypt0, COSE_Encrypt, COSE_Mac0, and COSE_Mac, including the multi-signer and multi-recipient variants. See [Message Types](#).
- 40 algorithms across signing, encryption, MAC, and key distribution (RFC 9053): ES256/384/512, EdDSA (Ed25519/Ed448), PS256/384/512, ML-DSA-44/65/87, AES-GCM (128/192/256), ChaCha20-Poly1305, AES-CCM variants, HMAC-SHA256/384/512, AES-MAC, Direct, AES Key Wrap, and ECDH-ES+HKDF. See [Algorithms](#).
- Standardized post-quantum signatures: ML-DSA (FIPS 204) at all three security levels, conformant to RFC 9964 ("ML-DSA for JOSE and COSE"). COSE keys use the RFC 9964 AKP key type (key 7) with a required alg, the public key in pub (-1), and the 32-byte seed private key in priv (-2).
- COSE_Key / COSE_KeySet serialization for all supported key types, including full RFC 8230 RSA private keys (n, e, d, p, q, dP, dQ, qInv).
- Zero dynamic allocation: every operation uses caller-provided buffers, with no heap, .data, or .bss usage.
- Path to FIPS 140-3 through wolfCrypt FIPS Certificate #4718 (sole crypto dependency).
- WOLFCOSE_LEAN configuration layer with WOLFCOSE_HAVE_* feature gates, WOLFCOSE_LEAN_VERIFY / ML-DSA lean profiles for verify-only targets, and a WOLFCOSE_MIN_BUFFERS bounded-stack profile. Verify-only ECC builds link against sign-disabled wolfCrypt (NO_ECC_SIGN, NO_ASN, no mp_int); the ECC signing helpers are gated out so a verify-only image never pulls in sign code, enforced in CI without -ffunction-sections garbage collection. See [Macros](#).
- LIBWOLFCOSE_VERSION_STRING / LIBWOLFCOSE_VERSION_HEX in wolfcose/version.h for compile-time version checks.

14.2.3 Fixes

- RSA private COSE_Key encode/decode now emits the RFC 8230 MUST-present dP (-6) and dQ (-7) CRT exponents and encodes d at full modulus width, so a private RSA key round-trips reliably against strict RSA decoders.
- COSE_Mac emits an empty protected header for direct-key recipients, matching the COSE structure other implementations expect on the wire.
- COSE_Key emits preferred (shortest) CBOR length for the RSA n and d byte strings, keeping serialized keys deterministic.

14.2.4 Improvements/Optimizations

- Minimal footprint: an ES256 COSE_Sign1 build is ~5.1 KB verify-only and ~6.8 KB sign + verify for the wolfCOSE COSE + CBOR engine. See [Footprint](#).
- MISRA C:2012 and C:2023 checked. See [MISRA Compliance](#).
- API hardening: COSE_Encrypt and wc_CoseMac_Create direct mode now require an explicit WOLFCOSE_ALG_DIRECT and reject a zero-initialized algorithm id; wc_CoseMac_Verify classifies the recipient algorithm and enforces the algId policy; the CBOR wc_CBOR_PeekType peek is guarded against NULL and end-of-buffer reads with a single-exit sentinel return; the wc_CoseSign1_Verify and symmetric COSE_Encrypt0 key parameters are const-qualified; and ephemeral COSE_Key curve ids are range-checked before any narrowing cast.
- Coverity DEADCODE findings in the COSE MAC and CBOR decode paths resolved; static analysis (cppcheck, Clang analyzer, GCC -fanalyzer, Coverity) is clean.
- CI matrix covering Ubuntu/macOS, GCC 10-14 and Clang 14-18, ~240 algorithm combination tests, static analysis (cppcheck, Clang analyzer, GCC -fanalyzer, Coverity), security scanning (CodeQL, Semgrep) and house-style gates, sanitizers (ASan/UBSan), a wolfCOSE <-> t_cose wire-interop conformance suite, and a wolfSSL version matrix with explicit ML-DSA/PQC rows. See

Testing.

- Expanded negative and boundary coverage: 4 KB large-payload round-trips for COSE_Encrypt0/COSE_Mac0, empty-payload round-trips across AES-GCM/AES-CCM/ChaCha20-Poly1305, CBOR integer argument-width boundaries through the 8-byte and INT64_MIN extremes with pinned encoded lengths, HMAC-384/512 short-key rejection, and pinned MAC tag lengths (including an AES-CBC-MAC block-boundary known-answer test) with IV-chaining tamper checks.

wolfCOSE 1.0.0 has been developed according to wolfSSL's development and QA process (see the [wolfSSL Software Development Process and Quality Assurance](#) page) and successfully passed the quality criteria.

Requires wolfSSL 5.8.0 or later as the crypto backend; ML-DSA support requires wolfSSL 5.9.2 or later. See [Getting Started](#) for build instructions.