Python CWT

Release 1.4.2

AJITOMI Daisuke

Oct 16, 2021

CONTENTS

Python CWT is a CBOR Web Token (CWT) and CBOR Object Signing and Encryption (COSE) implementation compliant with:

- RFC8392: CBOR Web Token (CWT)
- RFC8152: CBOR Object Signing and Encryption (COSE)
- and related various specifications. See Referenced Specifications.

It is designed to make users who already know about JWS/JWE/JWT be able to use it in ease. Little knowledge of CBOR/COSE/CWT is required to use it.

You can install Python CWT with pip:

```
$ pip install cwt
```

And then, you can use it as follows:

```
>>> import cwt
>>> from cwt import COSEKey
>>> key = COSEKey.from_symmetric_key(alg="HS256")
>>> token = cwt.encode({"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},␣
↪key)
>>> token.hex()

↪'d18443a10105a05835a60172636f6170733a2f2f61732e6578616d706c65026764616a69616a690743313233041a60c6a60b0
↪'
>>> cwt.decode(token, key)
{1: 'coaps://as.example', 2: 'dajiaji', 7: b'123', 4: 1620088759, 5: 1620085159, 6:␣
↪1620085159}
```

CHAPTER

ONE

INDEX

1.1 Installation

You can install Python CWT with pip:

```
$ pip install cwt
```

1.2 CWT Usage Examples

The following is a simple sample code using CWT API:

```
>>> import cwt
>>> from cwt import Claims, COSEKey
>>> key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
>>> token = cwt.encode({"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},␣
↪key)
>>> token.hex()

↪'d18443a10105a05835a60172636f6170733a2f2f61732e6578616d706c65026764616a69616a690743313233041a609097b7
↪'
>>> cwt.decode(token, key)
{1: 'coaps://as.example', 2: 'dajiaji', 7: b'123', 4: 1620088759, 5: 1620085159, 6:␣
↪1620085159}
```

This page shows various examples to use CWT API in this library.

- *MACed CWT*
- *Signed CWT*
- *Encrypted CWT*
- *Nested CWT*
- *CWT with User Settings*
- *CWT with User-Defined Claims*
- *CWT with PoP key*
- *CWT for EUDCC (EU Digital COVID Certificate)*

1.2.1 MACed CWT

Create a MACed CWT, verify and decode it as follows:

```
import cwt
from cwt import COSEKey

try:
    key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
    token = cwt.encode(
        {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},
        key,
    )
    decoded = cwt.decode(token, key)

    # If you want to treat the result like a JWT;
    readable = Claims.new(decoded)
    assert readable.iss == "coaps://as.example"
    assert readable.sub == "dajiaji"
    assert readable.cti == "123"
    # readable.exp == 1620088759
    # readable.nbf == 1620085159
    # readable.iat == 1620085159

except Exception as err:
    # All the other examples in this document omit error handling but this CWT library
    # can throw following errors:
    #   ValueError: Invalid arguments.
    #   EncodeError: Failed to encode.
    #   VerifyError: Failed to verify.
    #   DecodeError: Failed to decode.
    print(err)
```

A raw CWT structure (Dict[int, Any]) can also be used as follows:

```
import cwt
from cwt import COSEKey

key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
token = cwt.encode({1: "coaps://as.example", 2: "dajiaji", 7: b"123"}, key)
decoded = cwt.decode(token, key)
```

Algorithms other than `HS256` are listed in Supported COSE Algorithms .

1.2.2 Signed CWT

Create an Ed25519 (Ed25519 for use w/ EdDSA only) key pair:

```
$ openssl genpkey -algorithm ed25519 -out private_key.pem
$ openssl pkey -in private_key.pem -pubout -out public_key.pem
```

Create a Signed CWT, verify and decode it with the key pair as follows:

```
import cwt
from cwt import COSEKey

# The sender side:
with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="01")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

# The recipient side:
with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="01")
decoded = cwt.decode(token, public_key)
```

JWKs can also be used instead of the PEM-formatted keys as follows:

```
import cwt
from cwt import COSEKey

# The sender side:
private_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "key_ops": ["sign"],
        "alg": "EdDSA",
        "crv": "Ed25519",
        "x": "2E6dX83gqD_D0eAmqnaHe1TC1xuld6iAKXfw2OVATr0",
        "d": "L8JS08VsFZoZxGa9JvzYmCWOwg7zaKcei3KZmYsj7dc",
    }
)
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

# The recipient side:
public_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "key_ops": ["verify"],
        "crv": "Ed25519",
        "x": "2E6dX83gqD_D0eAmqnaHe1TC1xuld6iAKXfw2OVATr0",
    }
```

(continues on next page)

(continued from previous page)

```
)
decoded = cwt.decode(token, public_key)
```

Algorithms other than `Ed25519` are also supported. The following is an example of `ES256`:

```
$ openssl ecparam -genkey -name prime256v1 -noout -out private_key.pem
$ openssl ec -in private_key.pem -pubout -out public_key.pem
```

```
import cwt
from cwt import COSEKey

with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="01")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="01")
decoded = cwt.decode(token, public_key)
```

Other supported algorithms are listed in Supported COSE Algorithms.

1.2.3 Encrypted CWT

Create an encrypted CWT with `ChaCha20/Poly1305` (ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag), and decrypt it as follows:

```
import cwt
from cwt import COSEKey

enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305", kid="01")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, enc_key
)
decoded = cwt.decode(token, enc_key)
```

Algorithms other than `ChaCha20/Poly1305` are also supported. The following is an example of `AES-CCM-16-64-256`:

```
import cwt
from cwt import COSEKey

enc_key = COSEKey.from_symmetric_key(alg="AES-CCM-16-64-256", kid="01")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, enc_key
)
decoded = cwt.decode(token, enc_key)
```

Other supported algorithms are listed in Supported COSE Algorithms.

1.2.4 Nested CWT

Create a signed CWT and encrypt it, and then decrypt and verify the nested CWT as follows.

```
import cwt
from cwt import COSEKey

# A shared encryption key.
enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305", kid="enc-01")

# Creates a CWT with ES256 signing.
with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="sig-01")
token = cwt.encode(
    {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, private_key
)

# Encrypts the signed CWT.
nested = cwt.encode(token, enc_key)

# Decrypts and verifies the nested CWT.
with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="sig-01")
decoded = cwt.decode(nested, [enc_key, public_key])
```

1.2.5 CWT with User Settings

The `cwt` in `cwt.encode()` and `cwt.decode()` above is a global `CWT` class instance created with default settings in advance. The default settings are as follows:

- `expires_in`: `3600` seconds. This is the default lifetime in seconds of CWTs.
- `leeway`: `60` seconds. This is the default leeway in seconds for validating `exp` and `nbf`.

If you want to change the settings, you can create your own `CWT` class instance as follows:

```
from cwt import COSEKey, CWT

key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
mycwt = CWT.new(expires_in=3600 * 24, leeway=10)
token = mycwt.encode({"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"}, key)
decoded = mycwt.decode(token, key)
```

1.2.6 CWT with User-Defined Claims

You can use your own claims as follows:

Note that such user-defined claim's key should be less than -65536.

```
import cwt
from cwt import COSEKey

# The sender side:
```

(continues on next page)

(continued from previous page)

```
with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="01")
token = cwt.encode(
    {
        1: "coaps://as.example",  # iss
        2: "dajiaji",  # sub
        7: b"123",  # cti
        -70001: "foo",
        -70002: ["bar"],
        -70003: {"baz": "qux"},
        -70004: 123,
    },
    private_key,
)

# The recipient side:
with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="01")
raw = cwt.decode(token, public_key)
assert raw[-70001] == "foo"
assert raw[-70002][0] == "bar"
assert raw[-70003]["baz"] == "qux"
assert raw[-70004] == 123

readable = Claims.new(raw)
assert readable.get(-70001) == "foo"
assert readable.get(-70002)[0] == "bar"
assert readable.get(-70003)["baz"] == "qux"
assert readable.get(-70004) == 123
```

User-defined claims can also be used with JSON-based claims as follows:

```
import cwt
from cwt import Claims, COSEKey

with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="01")

my_claim_names = {
    "ext_1": -70001,
    "ext_2": -70002,
    "ext_3": -70003,
    "ext_4": -70004,
}

cwt.set_private_claim_names(my_claim_names)
token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": b"123",
        "ext_1": "foo",
```

(continues on next page)

(continued from previous page)

```
        "ext_2": ["bar"],
        "ext_3": {"baz": "qux"},
        "ext_4": 123,
    },
    private_key,
)
claims.set_private_claim_names()

with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="01")

raw = cwt.decode(token, public_key)
readable = Claims.new(
    raw,
    private_claim_names=my_claim_names,
)
assert readable.get("ext_1") == "foo"
assert readable.get("ext_2")[0] == "bar"
assert readable.get("ext_3")["baz"] == "qux"
assert readable.get("ext_4") == 123
```

1.2.7 CWT with PoP key

Create a CWT which has a PoP key as follows:

On the issuer side:

```
import cwt
from cwt import COSEKey

# Prepares a signing key for CWT in advance.
with open("./private_key_of_issuer.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="issuer-01")

# Sets the PoP key to a CWT for the presenter.
token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": "123",
        "cnf": {
            "jwk": {  # Provided by the CWT presenter.
                "kid": "presenter-01",
                "kty": "OKP",
                "use": "sig",
                "crv": "Ed25519",
                "x": "2E6dX83gqD_D0eAmqnaHe1TC1xuld6iAKXfw2OVATr0",
                "alg": "EdDSA",
            },
        },
    },
```

(continues on next page)

(continued from previous page)

```
    private_key,
)

# Issues the token to the presenter.
```

On the CWT presenter side:

```
import cwt
from cwt import COSEKey

# Prepares a private PoP key in advance.
with open("./private_pop_key.pem") as key_file:
    pop_key_private = COSEKey.from_pem(key_file.read(), kid="presenter-01")

# Receives a message (e.g., nonce)  from the recipient.
msg = b"could-you-sign-this-message?"  # Provided by recipient.

# Signs the message with the private PoP key.
sig = pop_key_private.sign(msg)

# Sends the msg and the sig with the CWT to the recipient.
```

On the CWT recipient side:

```
import cwt
from cwt import Claims, COSEKey

# Prepares the public key of the issuer in advance.
with open("./public_key_of_issuer.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="issuer-01")

# Verifies and decodes the CWT received from the presenter.
raw = cwt.decode(token, public_key)
decoded = Claims.new(raw)

# Extracts the PoP key from the CWT.
extracted_pop_key = COSEKey.new(decoded.cnf)  #  = raw[8][1]

# Then, verifies the message sent by the presenter
# with the signature which is also sent by the presenter as follows:
extracted_pop_key.verify(msg, sig)
```

In case of another PoP confirmation method `Encrypted_COSE_Key`:

```
import cwt
from cwt import Claims, COSEKey, EncryptedCOSEKey

with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="issuer-01")

enc_key = COSEKey.from_symmetric_key(
    "a-client-secret-of-cwt-recipient",  # Just 32 bytes!
    alg="ChaCha20/Poly1305",
```

(continues on next page)

(continued from previous page)

```
    kid="recipient-01",
)
pop_key = COSEKey.from_symmetric_key(
    "a-client-secret-of-cwt-presenter",
    alg="HMAC 256/256",
    kid="presenter-01",
)

token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": "123",
        "cnf": {
            # 'eck'(Encrypted Cose Key) is a keyword defined by this library.
            "eck": EncryptedCOSEKey.from_cose_key(pop_key, enc_key),
        },
    },
    private_key,
)

with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="issuer-01")
raw = cwt.decode(token, public_key)
decoded = Claims.new(raw)
extracted_pop_key = EncryptedCOSEKey.to_cose_key(decoded.cnf, enc_key)
# extracted_pop_key.verify(message, signature)
```

In case of another PoP confirmation method `kid`:

```
import cwt
from cwt import Claims, COSEKey

with open("./private_key.pem") as key_file:
    private_key = COSEKey.from_pem(key_file.read(), kid="issuer-01")

token = cwt.encode(
    {
        "iss": "coaps://as.example",
        "sub": "dajiaji",
        "cti": "123",
        "cnf": {
            "kid": "pop-key-id-of-cwt-presenter",
        },
    },
    private_key,
)

with open("./public_key.pem") as key_file:
    public_key = COSEKey.from_pem(key_file.read(), kid="issuer-01")
raw = cwt.decode(token, public_key)
decoded = Claims.new(raw)
```

(continues on next page)

(continued from previous page)

```
# decoded.cnf(=raw[8][3]) is kid.
```

1.2.8 CWT for EUDCC (EU Digital COVID Certificate)

Python CWT supports Electronic Health Certificate Specification and EUDCC (EU Digital COVID Certificate) compliant with Technical Specifications for Digital Green Certificates Volume 1.

A following example shows how to verify an EUDCC:

```
import cwt
from cwt import Claims, load_pem_hcert_dsc

# A DSC(Document Signing Certificate) issued by a CSCA
# (Certificate Signing Certificate Authority) quoted from:
# https://github.com/eu-digital-green-certificates/dgc-testdata/blob/main/AT/2DCode/raw/
↪1.json
dsc = "-----BEGIN CERTIFICATE-----\
↪nMIIBvTCCAWOgAwIBAgIKAXk8i88OleLsuTAKBggqhkjOPQQDAjA2MRYwFAYDVQQDDA1BVCBER0MgQ1NDQSAxMQswCQYDVQQGEwJB
↪M4nc=\n-----END CERTIFICATE-----"

# An EUDCC (EU Digital COVID Certificate) quoted from:
# https://github.com/eu-digital-green-certificates/dgc-testdata/blob/main/AT/2DCode/raw/
↪1.json
eudcc = bytes.fromhex(

↪"d2844da20448d919375fc1e7b6b20126a0590133a4041a61817ca0061a60942ea001624154390103a101a4617681aa62646e
↪"
)

public_key = load_pem_hcert_dsc(dsc)
decoded = cwt.decode(eudcc, keys=[public_key])
claims = Claims.new(decoded)
# claims.hcert[1] ==
# {
#     'v': [
#         {
#             'dn': 1,
#             'ma': 'ORG-100030215',
#             'vp': '1119349007',
#             'dt': '2021-02-18',
#             'co': 'AT',
#             'ci': 'URN:UVCI:01:AT:10807843F94AEE0EE5093FBC254BD813#B',
#             'mp': 'EU/1/20/1528',
#             'is': 'Ministry of Health, Austria',
#             'sd': 2,
#             'tg': '840539006',
#         }
#     ],
#     'nam': {
#         'fnt': 'MUSTERFRAU<GOESSINGER',
#         'fn': 'Musterfrau-Gößinger',
```

(continues on next page)

(continued from previous page)

```
#         'gnt': 'GABRIELE',
#         'gn': 'Gabriele',
#     },
#     'ver': '1.0.0',
#     'dob': '1998-02-26',
# }
```

1.3 COSE Usage Examples

The following is a simple sample code using COSE API:

```
>>> from cwt import COSE, COSEKey
>>> ctx = COSE.new(alg_auto_inclusion=True, kid_auto_inclusion=True)
>>> mac_key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
>>> encoded = ctx.encode_and_mac(b"Hello world!", mac_key)
>>> encoded.hex()

↪'d18443a10105a1044230314c48656c6c6f20776f726c642158205d0b144add282ccaac32a02e0d5eec76928ccadf3623271e
↪'
>>> ctx.decode(encoded, mac_key)
b'Hello world!'
```

This page shows various examples to use COSE API in this library.

1.3.1 COSE MAC0

Create a COSE MAC0 message, verify and decode it as follows:

```
from cwt import COSE, COSEKey

mac_key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
ctx = COSE.new(alg_auto_inclusion=True, kid_auto_inclusion=True)
encoded = ctx.encode_and_mac(b"Hello world!", mac_key)
assert b"Hello world!" == ctx.decode(encoded, mac_key)
```

Algorithms other than `HS256` are listed in Supported COSE Algorithms .

Following two samples are other ways of writing the above example:

```
from cwt import COSE, COSEKey

mac_key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
ctx = COSE.new()
encoded = ctx.encode_and_mac(
    b"Hello world!",
    mac_key,
    protected={"alg": "HS256"},
    unprotected={"kid": "01"},
)
assert b"Hello world!" == ctx.decode(encoded, mac_key)
```

```
from cwt import COSE, COSEKey

mac_key = COSEKey.from_symmetric_key(alg="HS256", kid="01")
ctx = COSE.new()
encoded = ctx.encode_and_mac(
    b"Hello world!",
    mac_key,
    protected={1: 5},
    unprotected={4: b"01"},
)
assert b"Hello world!" == ctx.decode(encoded, mac_key)
```

1.3.2 COSE MAC

Direct Key Distribution

The direct key distribution shares a MAC key between the sender and the recipient that is used directly. The follwing example shows the simplest way to make a COSE MAC message, verify and decode it with the direct key distribution method.

```
from cwt import COSE, COSEKey, Recipient

# The sender makes a COSE MAC message as follows:
mac_key = COSEKey.from_symmetric_key(alg="HS512", kid="01")
r = Recipient.from_jwk({"alg": "direct"})
```

(continues on next page)

(continued from previous page)

```
r.apply(mac_key)
ctx = COSE.new()
encoded = ctx.encode_and_mac(b"Hello world!", mac_key, recipients=[r])

# The recipient has the same MAC key and can verify and decode it:
assert b"Hello world!" == ctx.decode(encoded, mac_key)
```

Following samples are other ways of writing the above sample:

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
# In contrast to from_jwk(), new() is low-level constructor.
mac_key = COSEKey.from_symmetric_key(alg="HS512", kid="01")
r = Recipient.new(unprotected={"alg": "direct"})
r.apply(mac_key)
ctx = COSE.new()
encoded = ctx.encode_and_mac(b"Hello world!", mac_key, recipients=[r])

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, mac_key)
```

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
# new() can accept following raw COSE header parameters.
mac_key = COSEKey.from_symmetric_key(alg="HS512", kid="01")
r = Recipient.new(unprotected={1: 7})
r.apply(mac_key)
ctx = COSE.new()
encoded = ctx.encode_and_mac(b"Hello world!", mac_key, recipients=[r])

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, mac_key)
```

Direct Key with KDF

```
from secrets import token_bytes
from cwt import COSE, COSEKey, Recipient

shared_material = token_bytes(32)
shared_key = COSEKey.from_symmetric_key(shared_material, kid="01")

# The sender side:
r = Recipient.from_jwk(
    {
        "kty": "oct",
        "alg": "direct+HKDF-SHA-256",
    },
)
```

(continues on next page)

(continued from previous page)

```
mac_key = r.apply(shared_key, context={"alg": "HS256"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_mac(
    b"Hello world!",
    key=mac_key,
    recipients=[r],
)

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, shared_key, context={"alg": "HS256"})
```

AES Key Wrap

The AES key wrap algorithm can be used to wrap a MAC key as follows:

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
mac_key = COSEKey.from_symmetric_key(alg="HS512")
r = Recipient.from_jwk(
    {
        "kid": "01",
        "alg": "A128KW",
        "k": "hJtXIZ2uSN5kbQfbtTNWbg",  # A shared wrapping key
    },
)
r.apply(mac_key)
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_mac(b"Hello world!", key=mac_key, recipients=[r])

# The recipient side:
shared_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "oct",
        "alg": "A128KW",
        "k": "hJtXIZ2uSN5kbQfbtTNWbg",
    },
)
assert b"Hello world!" == ctx.decode(encoded, shared_key)
```

Direct Key Agreement

The direct key agreement methods can be used to create a shared secret. A KDF (Key Distribution Function) is then applied to the shared secret to derive a key to be used to protect the data. The follwing example shows a simple way to make a COSE Encrypt message, verify and decode it with the direct key agreement methods (`ECDH-ES+HKDF-256` with various curves).

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
r = Recipient.from_jwk(
    {
        "kty": "EC",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "P-256",
    },
)
# The following key is provided by the recipient in advance.
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
    }
)
mac_key = r.apply(recipient_key=pub_key, context={"alg": "HS256"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_mac(
    b"Hello world!",
    key=mac_key,
    recipients=[r],
)

# The recipient side:
# The following key is the private key of the above pub_key.
priv_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
        "d": "r_kHyZ-a06rmxM3yESK84r1otSg-aQcVStkRhA-iCM8",
    }
)
# The enc_key will be derived in decode() with priv_key and
# the sender's public key which is conveyed as the recipient
# information structure in the COSE Encrypt message (encoded).
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "HS256"})
```

You can use other curves (`P-384`, `P-521`, `X25519`, `X448`) instead of `P-256`:

In case of `X25519`:

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
r = Recipient.from_jwk(
    {
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X25519",
    },
)
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X25519",
        "x": "y3wJq3uXPHeoCO4FubvTc7VcBuqpvUrSvU6ZMbHDTCI",
    }
)
mac_key = r.apply(recipient_key=pub_key, context={"alg": "HS256"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_mac(
    b"Hello world!",
    key=mac_key,
    recipients=[r],
)

# The recipient side:
priv_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X25519",
        "x": "y3wJq3uXPHeoCO4FubvTc7VcBuqpvUrSvU6ZMbHDTCI",
        "d": "vsJ1oX5NNi0IGdwGldiac75r-Utmq3Jq4LGv48Q_Qc4",
    }
)
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "HS256"})
```

In case of `X448`:

```
from cwt import COSE, COSEKey, Recipient

r = Recipient.from_jwk(
    {
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X448",
    },
```

(continues on next page)

(continued from previous page)

```
)
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X448",
        "x": "IkLmc0klvEMXYneHMKAB6ePohryAwAPVe2pRSffIDY6NrjeYNWVX5J-fG4NV2OoU77C88A0mvxI
↪",
    }
)
mac_key = r.apply(recipient_key=pub_key, context={"alg": "HS256"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_mac(
    b"Hello world!",
    key=mac_key,
    recipients=[r],
)
priv_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X448",
        "x": "IkLmc0klvEMXYneHMKAB6ePohryAwAPVe2pRSffIDY6NrjeYNWVX5J-fG4NV2OoU77C88A0mvxI
↪",
        "d": "rJJRG3nshyCtd9CgXld8aNaB9YXKR0UOi7zj7hApg9YH4XdBO0G8NcAFNz_uPH2GnCZVcSDgV5c
↪",
    }
)
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "HS256"})
```

Key Agreement with Key Wrap

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
mac_key = COSEKey.from_symmetric_key(alg="HS256")
r = Recipient.from_jwk(
    {
        "kty": "EC",
        "alg": "ECDH-SS+A128KW",
        "crv": "P-256",
        "x": "7cvYCcdU22WCwW1tZXR8iuzJLWGcd46xfxO1XJs-SPU",
        "y": "DzhJXgz9RI6TseNmwEfLoNVns8UmvONsPzQDop2dKoo",
        "d": "Uqr4fay_qYQykwcNCB2efj_NFaQRRQ-6fHZm763jt5w",
    }
)
pub_key = COSEKey.from_jwk(
    {
```

(continues on next page)

(continued from previous page)

```
        "kid": "meriadoc.brandybuck@buckland.example",
        "kty": "EC",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
    }
)
r.apply(mac_key, recipient_key=pub_key, context={"alg": "HS256"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_mac(
    b"Hello world!",
    key=mac_key,
    recipients=[r],
)

# The recipient side:
priv_key = COSEKey.from_jwk(
    {
        "kid": "meriadoc.brandybuck@buckland.example",
        "kty": "EC",
        "alg": "ECDH-SS+A128KW",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
        "d": "r_kHyZ-a06rmxM3yESK84r1otSg-aQcVStkRhA-iCM8",
    }
)
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "HS256"})
```

1.3.3 COSE Encrypt0

Create a COSE Encrypt0 message, verify and decode it as follows:

```
from cwt import COSE, COSEKey

enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305", kid="01")

# The sender side:
nonce = enc_key.generate_nonce()
ctx = COSE.new(alg_auto_inclusion=True, kid_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(b"Hello world!", enc_key, nonce=nonce)

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, enc_key)
```

Algorithms other than `ChaCha20/Poly1305` are listed in Supported COSE Algorithms .

Following two samples are other ways of writing the above example:

```
from cwt import COSE, COSEKey
```

(continues on next page)

(continued from previous page)

```
enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305", kid="01")

# The sender side:
nonce = enc_key.generate_nonce()
ctx = COSE.new()
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    enc_key,
    nonce=nonce,
    protected={"alg": "ChaCha20/Poly1305"},
    unprotected={"kid": "01"},
)

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, enc_key)
```

```
from cwt import COSE, COSEKey

enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305", kid="01")

# The sender side:
nonce = enc_key.generate_nonce()
ctx = COSE.new()
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    enc_key,
    nonce=nonce,
    protected={1: 24},
    unprotected={4: b"01"},
)

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, enc_key)
```

1.3.4 COSE Encrypt

Direct Key Distribution

The direct key distribution shares an encryption key between the sender and the recipient that is used directly. The follwing example shows the simplest way to make a COSE Encrypt message, verify and decode it with the direct key distribution method.

```
from cwt import COSE, COSEKey, Recipient

enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305", kid="01")

# The sender side:
nonce = enc_key.generate_nonce()
r = Recipient.from_jwk({"alg": "direct"})
r.apply(enc_key)
```

(continues on next page)

(continued from previous page)

```
ctx = COSE.new()
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    enc_key,
    nonce=nonce,
    recipients=[r],
)

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, enc_key)
```

Direct Key with KDF

```
from cwt import COSE, COSEKey, Recipient

shared_material = token_bytes(32)
shared_key = COSEKey.from_symmetric_key(shared_material, kid="01")

# The sender side:
r = Recipient.from_jwk(
    {
        "kty": "oct",
        "alg": "direct+HKDF-SHA-256",
    },
)
enc_key = r.apply(shared_key, context={"alg": "A256GCM"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    key=enc_key,
    recipients=[r],
)
# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, shared_key, context={"alg": "A256GCM"})
```

AES Key Wrap

The AES key wrap algorithm can be used to wrap an encryption key as follows:

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
r = Recipient.from_jwk(
    {
        "kid": "01",
        "kty": "oct",
        "alg": "A128KW",
        "k": "hJtXIZ2uSN5kbQfbtTNWbg",  # A shared wrapping key
    },
```

(continues on next page)

(continued from previous page)

```
)
enc_key = COSEKey.from_symmetric_key(alg="ChaCha20/Poly1305")
r.apply(enc_key)
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(b"Hello world!", key=enc_key, recipients=[r])

# The recipient side:
shared_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "oct",
        "alg": "A128KW",
        "k": "hJtXIZ2uSN5kbQfbtTNWbg",
    },
)
assert b"Hello world!" == ctx.decode(encoded, shared_key)
```

Direct Key Agreement

The direct key agreement methods can be used to create a shared secret. A KDF (Key Distribution Function) is then applied to the shared secret to derive a key to be used to protect the data. The follwing example shows a simple way to make a COSE Encrypt message, verify and decode it with the direct key agreement methods (`ECDH-ES+HKDF-256` with various curves).

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
r = Recipient.from_jwk(
    {
        "kty": "EC",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "P-256",
    },
)
# The following key is provided by the recipient in advance.
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
    }
)
enc_key = r.apply(recipient_key=pub_key, context={"alg": "A128GCM"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    key=enc_key,
    recipients=[r],
```

(continues on next page)

(continued from previous page)

```
)

# The recipient side:
# The following key is the private key of the above pub_key.
priv_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
        "d": "r_kHyZ-a06rmxM3yESK84r1otSg-aQcVStkRhA-iCM8",
    }
)
# The enc_key will be derived in decode() with priv_key and
# the sender's public key which is conveyed as the recipient
# information structure in the COSE Encrypt message (encoded).
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "A128GCM"})
```

You can use other curves (`P-384`, `P-521`, `X25519`, `X448`) instead of `P-256`:

In case of `X25519`:

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
r = Recipient.from_jwk(
    {
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X25519",
    },
)
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X25519",
        "x": "y3wJq3uXPHeoCO4FubvTc7VcBuqpvUrSvU6ZMbHDTCI",
    }
)
enc_key = r.apply(recipient_key=pub_key, context={"alg": "A128GCM"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    key=enc_key,
    recipients=[r],
)

# The recipient side:
priv_key = COSEKey.from_jwk(
```

(continues on next page)

(continued from previous page)

```
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X25519",
        "x": "y3wJq3uXPHeoCO4FubvTc7VcBuqpvUrSvU6ZMbHDTCI",
        "d": "vsJ1oX5NNi0IGdwGldiac75r-Utmq3Jq4LGv48Q_Qc4",
    }
)
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "A128GCM"})
```

In case of `X448`:

```
from cwt import COSE, COSEKey, Recipient

r = Recipient.from_jwk(
    {
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X448",
    },
)
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X448",
        "x": "IkLmc0klvEMXYneHMKAB6ePohryAwAPVe2pRSffIDY6NrjeYNWVX5J-fG4NV2OoU77C88A0mvxI
↪",
    }
)
enc_key = r.apply(recipient_key=pub_key, context={"alg": "A128GCM"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    key=enc_key,
    recipients=[r],
)
priv_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "OKP",
        "alg": "ECDH-ES+HKDF-256",
        "crv": "X448",
        "x": "IkLmc0klvEMXYneHMKAB6ePohryAwAPVe2pRSffIDY6NrjeYNWVX5J-fG4NV2OoU77C88A0mvxI
↪",
        "d": "rJJRG3nshyCtd9CgXld8aNaB9YXKR0UOi7zj7hApg9YH4XdBO0G8NcAFNz_uPH2GnCZVcSDgV5c
↪",
    }
)
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "A128GCM"})
```

Key Agreement with Key Wrap

```
from cwt import COSE, COSEKey, Recipient

# The sender side:
enc_key = COSEKey.from_symmetric_key(alg="A128GCM")
nonce = enc_key.generate_nonce()
r = Recipient.from_jwk(
    {
        "kty": "EC",
        "alg": "ECDH-SS+A128KW",
        "crv": "P-256",
        "x": "7cvYCcdU22WCwW1tZXR8iuzJLWGcd46xfxO1XJs-SPU",
        "y": "DzhJXgz9RI6TseNmwEfLoNVns8UmvONsPzQDop2dKoo",
        "d": "Uqr4fay_qYQykwcNCB2efj_NFaQRRQ-6fHZm763jt5w",
    }
)
pub_key = COSEKey.from_jwk(
    {
        "kid": "meriadoc.brandybuck@buckland.example",
        "kty": "EC",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
    }
)
r.apply(enc_key, recipient_key=pub_key, context={"alg": "A128GCM"})
ctx = COSE.new(alg_auto_inclusion=True)
encoded = ctx.encode_and_encrypt(
    b"Hello world!",
    key=enc_key,
    nonce=nonce,
    recipients=[r],
)

# The recipient side:
priv_key = COSEKey.from_jwk(
    {
        "kid": "meriadoc.brandybuck@buckland.example",
        "kty": "EC",
        "alg": "ECDH-SS+A128KW",
        "crv": "P-256",
        "x": "Ze2loSV3wrroKUN_4zhwGhCqo3Xhu1td4QjeQ5wIVR0",
        "y": "HlLtdXARY_f55A3fnzQbPcm6hgr34Mp8p-nuzQCE0Zw",
        "d": "r_kHyZ-a06rmxM3yESK84r1otSg-aQcVStkRhA-iCM8",
    }
)
assert b"Hello world!" == ctx.decode(encoded, priv_key, context={"alg": "A128GCM"})
```

1.3.5 COSE Signature1

Create a COSE Signature1 message, verify and decode it as follows:

```
from cwt import COSE, COSEKey

# The sender side:
priv_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
        "d": "V8kgd2ZBRuh2dgyVINBUqpPDr7BOMGcF22CQMIUHtNM",
    }
)
ctx = COSE.new(alg_auto_inclusion=True, kid_auto_inclusion=True)
encoded = ctx.encode_and_sign(b"Hello world!", priv_key)

# The recipient side:
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
    }
)
assert b"Hello world!" == ctx.decode(encoded, pub_key)
```

Following two samples are other ways of writing the above example:

```
from cwt import COSE, COSEKey

# The sender side:
sig_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
        "d": "V8kgd2ZBRuh2dgyVINBUqpPDr7BOMGcF22CQMIUHtNM",
    }
)
ctx = COSE.new()
encoded = ctx.encode_and_sign(
    b"Hello world!",
    sig_key,
    protected={"alg": "ES256"},
    unprotected={"kid": "01"},
)
```

(continues on next page)

(continued from previous page)

```
# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, sig_key)
```

```
from cwt import COSE, COSEKey

# The sender side:
sig_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
        "d": "V8kgd2ZBRuh2dgyVINBUqpPDr7BOMGcF22CQMIUHtNM",
    }
)
ctx = COSE.new()
encoded = ctx.encode_and_sign(
    b"Hello world!",
    sig_key,
    protected={1: -7},
    unprotected={4: b"01"},
)

# The recipient side:
assert b"Hello world!" == ctx.decode(encoded, sig_key)
```

1.3.6 COSE Signature

Create a COSE Signature message, verify and decode it as follows:

```
from cwt import COSE, COSEKey, Signer

# The sender side:
signer = Signer.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
        "d": "V8kgd2ZBRuh2dgyVINBUqpPDr7BOMGcF22CQMIUHtNM",
    },
)
ctx = COSE.new()
encoded = ctx.encode_and_sign(b"Hello world!", signers=[signer])

# The recipient side:
pub_key = COSEKey.from_jwk(
```

(continues on next page)

(continued from previous page)

```
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
    }
)
assert b"Hello world!" == ctx.decode(encoded, pub_key)
```

Following two samples are other ways of writing the above example:

```
from cwt import COSE, COSEKey, Signer

# The sender side:
signer = Signer.new(
    cose_key=COSEKey.from_jwk(
        {
            "kid": "01",
            "kty": "EC",
            "crv": "P-256",
            "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
            "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
            "d": "V8kgd2ZBRuh2dgyVINBUqpPDr7BOMGcF22CQMIUHtNM",
        }
    ),
    protected={"alg": "ES256"},
    unprotected={"kid": "01"},
)
ctx = COSE.new()
encoded = ctx.encode_and_sign(b"Hello world!", signers=[signer])

# The recipient side:
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
    }
)
assert b"Hello world!" == ctx.decode(encoded, pub_key)
```

```
from cwt import COSE, COSEKey, Signer

# The sender side:
signer = Signer.new(
    cose_key=COSEKey.from_jwk(
        {
            "kid": "01",
            "kty": "EC",
```

(continues on next page)

(continued from previous page)

```
            "crv": "P-256",
            "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
            "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
            "d": "V8kgd2ZBRuh2dgyVINBUqpPDr7BOMGcF22CQMIUHtNM",
        }
    ),
    protected={1: -7},
    unprotected={4: b"01"},
)
ctx = COSE.new()
encoded = ctx.encode_and_sign(b"Hello world!", signers=[signer])

# The recipient side:
pub_key = COSEKey.from_jwk(
    {
        "kid": "01",
        "kty": "EC",
        "crv": "P-256",
        "x": "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
        "y": "IBOL-C3BttVivg-lSreASjpkttcsz-1rb7btKLv8EX4",
    }
)
assert b"Hello world!" == ctx.decode(encoded, pub_key)
```

1.4 API Reference

A Python implementation of CWT/COSE <https://python-cwt.readthedocs.io>

cwt.encode(*claims: Union[*cwt.claims.Claims*, Dict[str, Any], Dict[int, Any], bytes]*, *key:* cwt.cose_key_interface.COSEKeyInterface, *nonce: bytes = b''*, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *signers: List[*cwt.signer.Signer*] = []*, *tagged: bool = False*) → bytes

cwt.encode_and_mac(*claims: Union[*cwt.claims.Claims*, Dict[int, Any], bytes]*, *key:* cwt.cose_key_interface.COSEKeyInterface, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *tagged: bool = False*) → bytes

cwt.encode_and_sign(*claims: Union[*cwt.claims.Claims*, Dict[int, Any], bytes]*, *key: Optional[*cwt.cose_key_interface.COSEKeyInterface*] = None*, *signers: List[*cwt.signer.Signer*] = []*, *tagged: bool = False*) → bytes

cwt.encode_and_encrypt(*claims: Union[*cwt.claims.Claims*, Dict[int, Any], bytes]*, *key:* cwt.cose_key_interface.COSEKeyInterface, *nonce: bytes = b''*, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *tagged: bool = False*) → bytes

cwt.decode(*data: bytes*, *keys: Union[*cwt.cose_key_interface.COSEKeyInterface*, List[*cwt.cose_key_interface.COSEKeyInterface*]]*, *no_verify: bool = False*) → Union[Dict[int, Any], bytes]

cwt.set_private_claim_names(*claim_names: Dict[str, int]*)

class cwt.**CWT**(*expires_in: int = 3600*, *leeway: int = 60*, *ca_certs: str = ''*)

Bases: cwt.cbor_processor.CBORProcessor

A CWT (CBOR Web Token) Implementaion, which is built on top of *COSE*

cwt.cwt is a global object of this class initialized with default settings.

CBOR_TAG = 61

classmethod new(*expires_in: int = 3600*, *leeway: int = 60*, *ca_certs: str = ''*)

Constructor.

Parameters

- **expires_in** (*int*) – The default lifetime in seconds of CWT (default value: 3600).
- **leeway** (*int*) – The default leeway in seconds for validating exp and nbf (default value: 60).
- **ca_certs** (*str*) – The path to a file which contains a concatenated list of trusted root certificates. You should specify private CA certificates in your target system. There should be no need to use the public CA certificates for the Web PKI.

Examples

```
>>> from cwt import CWT, COSEKey
>>> ctx = CWT.new(expires_in=3600*24, leeway=10)
>>> key = COSEKey.from_symmetric_key(alg="HS256")
>>> token = ctx.encode(
...     {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},
...     key,
... )
```

```
>>> from cwt import CWT, COSEKey
>>> ctx = CWT.new(expires_in=3600*24, leeway=10, ca_certs="/path/to/ca_certs")
>>> key = COSEKey.from_pem(alg="ES256")
>>> token = ctx.encode(
...     {"iss": "coaps://as.example", "sub": "dajiaji", "cti": "123"},
...     key,
... )
```

property expires_in: int

The default lifetime in seconds of CWT. If *exp* is not found in claims, this value will be used with current time.

property leeway: int

The default leeway in seconds for validating exp and nbf.

property cose: *cwt.cose.COSE*

The underlying COSE object.

encode(*claims: Union[cwt.claims.Claims, Dict[str, Any], Dict[int, Any], bytes]*, *key: cwt.cose_key_interface.COSEKeyInterface*, *nonce: bytes = b''*, *recipients: Optional[List[cwt.recipient_interface.RecipientInterface]] = None*, *signers: List[cwt.signer.Signer] = []*, *tagged: bool = False*) → bytes

Encodes CWT with MAC, signing or encryption. This is a wrapper function of the following functions for easy use:

- *encode_and_mac*

- *encode_and_sign*
- *encode_and_encrypt*

Therefore, it must be clear whether the use of the specified key is for MAC, signing, or encryption. For this purpose, the key must have the `key_ops` parameter set to identify the usage.

Parameters

- **claims** (*Union[Claims, Dict[str, Any], Dict[int, Any], bytes]*) – A CWT claims object, or a JWT claims object, text string or byte string.
- **key** (COSEKeyInterface) – A COSE key used to generate a MAC for the claims.
- **nonce** (*bytes*) – A nonce for encryption.
- **recipients** (*Optional[List[RecipientInterface]]*) – A list of recipient information structures.
- **signers** (*List[Signer]*) – A list of signer information structures for multiple signer cases.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

encode_and_mac(*claims: Union[*cwt.claims.Claims*, Dict[int, Any], bytes]*, *key:* cwt.cose_key_interface.COSEKeyInterface, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *tagged: bool = False*) → bytes

Encodes with MAC.

Parameters

- **claims** (*Union[Claims, Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (COSEKeyInterface) – A COSE key used to generate a MAC for the claims.
- **recipients** (*Optional[List[RecipientInterface]]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

encode_and_sign(*claims: Union[*cwt.claims.Claims*, Dict[int, Any], bytes]*, *key: Optional[*cwt.cose_key_interface.COSEKeyInterface*] = None*, *signers: List[*cwt.signer.Signer*] = []*, *tagged: bool = False*) → bytes

Encodes CWT with signing.

Parameters

- **claims** (*Claims*, *Union[Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (*Optional[COSEKeyInterface]*) – A COSE key or a list of the keys used to sign claims. When the `signers` parameter is set, this `key` parameter will be ignored and should not be set.
- **signers** (*List[Signer]*) – A list of signer information structures for multiple signer cases.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

encode_and_encrypt(*claims: Union[*cwt.claims.Claims*, Dict[int, Any], bytes]*, *key:* cwt.cose_key_interface.COSEKeyInterface, *nonce: bytes = b''*, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *tagged: bool = False*) → bytes

Encodes CWT with encryption.

Parameters

- **claims** (*Claims*, *Union[Dict[int, Any], bytes]*) – A CWT claims object or byte string.
- **key** (*COSEKeyInterface*) – A COSE key used to encrypt the claims.
- **nonce** (*bytes*) – A nonce for encryption.
- **recipients** (*List[RecipientInterface]*) – A list of recipient information structures.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A byte string of the encoded CWT.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode the claims.

decode(*data: bytes*, *keys: Union[*cwt.cose_key_interface.COSEKeyInterface*, List[*cwt.cose_key_interface.COSEKeyInterface*]]*, *no_verify: bool = False*) → Union[Dict[int, Any], bytes]

Verifies and decodes CWT.

Parameters

- **data** (*bytes*) – A byte string of an encoded CWT.
- **keys** (*Union[COSEKeyInterface, List[COSEKeyInterface]]*) – A COSE key or a list of the keys used to verify and decrypt the encoded CWT.
- **no_verify** (*bool*) – An indicator whether token verification is skiped or not.

Returns A byte string of the decoded CWT.

Return type Union[Dict[int, Any], bytes]

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the CWT.
- ***VerifyError*** – Failed to verify the CWT.

set_private_claim_names(*claim_names: Dict[str, int]*)

Sets private claim definitions. The definitions will be used in *encode* when it is called with JSON-based claims.

Parameters **claim_names** (*Dict[str, int]*) – A set of private claim definitions which consist of a readable claim name(str) and a claim key(int). The claim key should be less than -65536 but you can use the numbers other than pre-registered numbers listed in IANA Registry.

Raises **ValueError** – Invalid arguments.

class cwt.**COSE**(*alg_auto_inclusion: bool = False*, *kid_auto_inclusion: bool = False*, *verify_kid: bool = False*, *ca_certs: str = ''*)

Bases: cwt.cbor_processor.CBORProcessor

A COSE (CBOR Object Signing and Encryption) Implementaion built on top of cbor2.

classmethod new(*alg_auto_inclusion: bool = False*, *kid_auto_inclusion: bool = False*, *verify_kid: bool = False*, *ca_certs: str = ''*)

Constructor.

Parameters

- **alg_auto_inclusion** (*bool*) – The indicator whether alg parameter is included in a proper header bucket automatically or not.
- **kid_auto_inclusion** (*bool*) – The indicator whether kid parameter is included in a proper header bucket automatically or not.
- **verify_kid** (*bool*) – The indicator whether kid verification is mandatory or not.
- **ca_certs** (*str*) – The path to a file which contains a concatenated list of trusted root certificates. You should specify private CA certificates in your target system. There should be no need to use the public CA certificates for the Web PKI.

property alg_auto_inclusion: bool

If this property is True, an encode_and_*() function will automatically set the alg parameter in the header from the COSEKey argument.

property kid_auto_inclusion: bool

If this property is True, an encode_and_*() function will automatically set the kid parameter in the header from the COSEKey argument.

property verify_kid: bool

If this property is True, the decode() function will perform the verification and decoding process only if the kid of the COSE data to be decoded and one of the kid s in the key list given as an argument match exact.

encode_and_mac(*payload: bytes*, *key:* cwt.cose_key_interface.COSEKeyInterface, *protected: Optional[Union[dict, bytes]] = None*, *unprotected: Optional[dict] = None*, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *external_aad: bytes = b''*, *out: str = ''*) → Union[bytes, _cbor2.CBORTag]

Encodes data with MAC.

Parameters

- **payload** (*bytes*) – A content to be MACed.
- **key** (COSEKeyInterface) – A COSE key as a MAC Authentication key.
- **protected** (*Optional[Union[dict, bytes]]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*Optional[dict]*) – Parameters that are not cryptographically protected.
- **recipients** (*Optional[List[RecipientInterface]]*) – A list of recipient information structures.
- **external_aad** (*bytes*) – External additional authenticated data supplied by application.
- **out** (*str*) – An output format. Only "cbor2/CBORTag" can be used. If "cbor2/CBORTag" is specified. This function will return encoded data as cbor2's CBORTag object. If any other value is specified, it will return encoded data as bytes.

Returns A byte string of the encoded COSE or a cbor2.CBORTag object.

Return type Union[bytes, CBORTag]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode data.

encode_and_sign(*payload: bytes*, *key: Optional[cwt.cose_key_interface.COSEKeyInterface] = None*, *protected: Optional[Union[dict, bytes]] = None*, *unprotected: Optional[dict] = None*, *signers: List[cwt.signer.Signer] = []*, *external_aad: bytes = b''*, *out: str = ''*) → Union[bytes, _cbor2.CBORTag]

Encodes data with signing.

Parameters

- **payload** (*bytes*) – A content to be signed.
- **key** (*Optional[COSEKeyInterface]*) – A signing key for single signer cases. When the signers parameter is set, this key will be ignored and should not be set.
- **protected** (*Optional[Union[dict, bytes]]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*Optional[dict]*) – Parameters that are not cryptographically protected.
- **signers** (*List[Signer]*) – A list of signer information objects for multiple signer cases.
- **external_aad** (*bytes*) – External additional authenticated data supplied by application.
- **out** (*str*) – An output format. Only "cbor2/CBORTag" can be used. If "cbor2/CBORTag" is specified. This function will return encoded data as cbor2's CBORTag object. If any other value is specified, it will return encoded data as bytes.

Returns

A byte string of the encoded COSE or a cbor2.CBORTag object.

Return type Union[bytes, CBORTag]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode data.

encode_and_encrypt(*payload: bytes*, *key:* cwt.cose_key_interface.COSEKeyInterface, *protected: Optional[Union[dict, bytes]] = None*, *unprotected: Optional[dict] = None*, *nonce: bytes = b''*, *recipients: Optional[List[*cwt.recipient_interface.RecipientInterface*]] = None*, *external_aad: bytes = b''*, *out: str = ''*) → bytes

Encodes data with encryption.

Parameters

- **payload** (*bytes*) – A content to be encrypted.
- **key** (COSEKeyInterface) – A COSE key as an encryption key.
- **protected** (*Optional[Union[dict, bytes]]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*Optional[dict]*) – Parameters that are not cryptographically protected.
- **nonce** (*bytes*) – A nonce for encryption.
- **recipients** (*Optional[List[*RecipientInterface*]]*) – A list of recipient information structures.
- **external_aad** (*bytes*) – External additional authenticated data supplied by application.
- **out** (*str*) – An output format. Only "cbor2/CBORTag" can be used. If "cbor2/CBORTag" is specified. This function will return encoded data as cbor2's CBORTag object. If any other value is specified, it will return encoded data as bytes.

Returns

A byte string of the encoded COSE or a cbor2.CBORTag object.

Return type Union[bytes, CBORTag]

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode data.

decode(*data: Union[bytes, _cbor2.CBORTag]*, *keys: Union[*cwt.cose_key_interface.COSEKeyInterface, *List[*cwt.cose_key_interface.COSEKeyInterface*]]*, *context: Optional[Union[List[Any], Dict[str, Any]]] = None*, *external_aad: bytes = b''*) → bytes

Verifies and decodes COSE data.

Parameters

- **data** (*Union[bytes, CBORTag]*) – A byte string or cbor2.CBORTag of an encoded data.
- **keys** (*Union[*COSEKeyInterface, *List[*COSEKeyInterface*]]*) – COSE key(s) to verify and decrypt the encoded data.
- **context** (*Optional[Union[Dict[str, Any], List[Any]]]*) – A context information structure for key deriviation functions.
- **external_aad** (*bytes*) – External additional authenticated data supplied by application.

Returns A byte string of decoded payload.

Return type bytes

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode data.

- *VerifyError* – Failed to verify data.

class cwt.COSEKey
Bases: object

A COSEKeyInterface Builder.

static new(*params: Dict[int, Any]*) → *cwt.cose_key_interface.COSEKeyInterface*
Creates a COSE key from a CBOR-like dictionary with numeric keys.

Parameters **params** (*Dict[int, Any]*) – A CBOR-like dictionary with numeric keys of a COSE key.

Returns A COSE key object.

Return type *COSEKeyInterface*

Raises **ValueError** – Invalid arguments.

classmethod from_symmetric_key(*key: Union[bytes, str] = b''*, *alg: Union[int, str] = ''*, *kid: Union[bytes, str] = b''*, *key_ops: Optional[Union[List[int], List[str]]] = None*) → *cwt.cose_key_interface.COSEKeyInterface*
Creates a COSE key from a symmetric key.

Parameters

- **key** (*Union[bytes, str]*) – A key bytes or string.
- **alg** (*Union[int, str]*) – An algorithm label(int) or name(str). Supported alg are listed in Supported COSE Algorithms.
- **kid** (*Union[bytes, str]*) – A key identifier.
- **key_ops** (*Union[List[int], List[str]]*) – A list of key operation values. Following values can be used: 1("sign"), 2("verify"), 3("encrypt"), 4("decrypt"), 5("wrap key"), 6("unwrap key"), 7("derive key"), 8("derive bits"), 9("MAC create"), 10("MAC verify")

Returns A COSE key object.

Return type *COSEKeyInterface*

Raises **ValueError** – Invalid arguments.

classmethod from_bytes(*key_data: bytes*) → *cwt.cose_key_interface.COSEKeyInterface*
Creates a COSE key from CBOR-formatted key data.

Parameters **key_data** (*bytes*) – CBOR-formatted key data.

Returns A COSE key object.

Return type *COSEKeyInterface*

Raises

- **ValueError** – Invalid arguments.
- *DecodeError* – Failed to decode the key data.

classmethod from_jwk(*data: Union[str, bytes, Dict[str, Any]]*) → *cwt.cose_key_interface.COSEKeyInterface*
Creates a COSE key from JWK (JSON Web Key).

Parameters **jwk** (*Union[str, bytes, Dict[str, Any]]*) – JWK-formatted key data.

Returns A COSE key object.

Return type *COSEKeyInterface*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

classmethod from_pem(*key_data: Union[str, bytes]*, *alg: Union[int, str] = ''*, *kid: Union[bytes, str] = b''*, *key_ops: Optional[Union[List[int], List[str]]] = None*) → *cwt.cose_key_interface.COSEKeyInterface*

Creates a COSE key from PEM-formatted key data.

Parameters

- **key_data** (*bytes*) – A PEM-formatted key data.
- **alg** (*Union[int, str]*) – An algorithm label(int) or name(str). Different from ::func::*cwt.COSEKey.from_symmetric_key*, it is only used when an algorithm cannot be specified by the PEM data, such as RSA family algorithms.
- **kid** (*Union[bytes, str]*) – A key identifier.
- **key_ops** (*Union[List[int], List[str]]*) – A list of key operation values. Following values can be used: `1("sign")`, `2("verify")`, `3("encrypt")`, `4("decrypt")`, `5("wrap key")`, `6("unwrap key")`, `7("derive key")`, `8("derive bits")`, `9("MAC create")`, `10("MAC verify")`

Returns A COSE key object.

Return type *COSEKeyInterface*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

class cwt.**EncryptedCOSEKey**

Bases: `cwt.cbor_processor.CBORProcessor`

An encrypted COSE key.

static from_cose_key(*key:* cwt.cose_key_interface.COSEKeyInterface, *encryption_key:* cwt.cose_key_interface.COSEKeyInterface, *nonce: bytes = b''*, *tagged: bool = False*) → Union[List[Any], bytes]

Returns an encrypted COSE key formatted to COSE_Encrypt0 structure.

Parameters

- **key** – COSEKeyInterface: A key to be encrypted.
- **encryption_key** – COSEKeyInterface: An encryption key to encrypt the target COSE key.
- **nonce** (*bytes*) – A nonce for encryption.
- **tagged** (*bool*) – An indicator whether the response is wrapped by CWT tag(61) or not.

Returns A COSE_Encrypt0 structure of the target COSE key.

Return type Union[List[Any], bytes]

Raises

- **ValueError** – Invalid arguments.

- **EncodeError** – Failed to encrypt the COSE key.

static to_cose_key(*key: List[Any]*, *encryption_key:* cwt.cose_key_interface.COSEKeyInterface) → *cwt.cose_key_interface.COSEKeyInterface*

Returns an decrypted COSE key.

Parameters

- **key** – COSEKeyInterface: A key formatted to COSE_Encrypt0 structure to be decrypted.
- **encryption_key** – COSEKeyInterface: An encryption key to decrypt the target COSE key.

Returns A key decrypted.

Return type *COSEKeyInterface*

Raises

- **ValueError** – Invalid arguments.
- **DecodeError** – Failed to decode the COSE key.
- **VerifyError** – Failed to verify the COSE key.

class cwt.Claims(*claims: Dict[int, Any]*, *claim_names: Dict[str, int] = {'EAT-FDO': - 257, 'EATMAROEPrefix': - 258, 'EUPHNonce': - 259, 'aud': 3, 'cnf': 8, 'cti': 7, 'dbgstat': 16, 'eat_profile': 18, 'exp': 4, 'hcert': - 260, 'iat': 6, 'iss': 1, 'location': 17, 'nbf': 5, 'nonce': 10, 'oemid': 13, 'secboot': 15, 'seclevel': 14, 'sub': 2, 'submods': 20, 'ueid': 11}*)

Bases: object

A class for handling CWT Claims like JWT claims.

classmethod new(*claims: Dict[int, Any]*, *private_claim_names: Dict[str, int] = {}*)

Creates a Claims object from a CBOR-like(Dict[int, Any]) claim object.

Parameters

- **claims** (*Dict[str, Any]*) – A CBOR-like(Dict[int, Any]) claim object.
- **private_claim_names** (*Dict[str, int]*) – A set of private claim definitions which consist of a readable claim name(str) and a claim key(int). The claim key should be less than -65536 but you can use the numbers other than pre-registered numbers listed in IANA Registry.

Returns A CWT claims object.

Return type *Claims*

Raises **ValueError** – Invalid arguments.

classmethod from_json(*claims: Union[str, bytes, Dict[str, Any]]*, *private_claim_names: Dict[str, int] = {}*)

Converts a JWT claims object into a CWT claims object which has numeric keys. If a key string in JSON data cannot be mapped to a numeric key, it will be skipped.

Parameters

- **claims** (*Union[str, bytes, Dict[str, Any]]*) – A JWT claims object to be converted.
- **private_claim_names** (*Dict[str, int]*) – A set of private claim definitions which consist of a readable claim name(str) and a claim key(int). The claim key should be less than -65536 but you can use the numbers other than pre-registered numbers listed in IANA Registry.

Returns A CWT claims object.

Return type *Claims*

Raises **ValueError** – Invalid arguments.

classmethod validate(*claims: Dict[int, Any]*)
Validates a CWT claims object.

Parameters **claims** (*Dict[int, Any]*) – A CWT claims object to be validated.

Raises **ValueError** – Failed to verify.

property iss: Optional[str]

property sub: Optional[str]

property aud: Optional[str]

property exp: Optional[int]

property nbf: Optional[int]

property iat: Optional[int]

property cti: Optional[str]

property hcert: Optional[dict]

property cnf: Optional[Union[Dict[int, Any], List[Any], str]]

get(*key: Union[str, int]*) → Any
Gets a claim value with a claim key.

Parameters **key** (*Union[str, int]*) – A claim key.

Returns The value of the claim.

Return type Any

to_dict() → Dict[int, Any]
Returns a raw claim object.

Returns The value of the raw claim.

Return type Any

class cwt.**Recipient**
Bases: object

A RecipientInterface Builder.

classmethod new(*protected: dict = {}*, *unprotected: dict = {}*, *ciphertext: bytes = b''*, *recipients: List[Any] = []*, *sender_key: Optional[*cwt.cose_key_interface.COSEKeyInterface*] = None*) → *cwt.recipient_interface.RecipientInterface*
Creates a recipient from a CBOR-like dictionary with numeric keys.

Parameters

- **protected** (*dict*) – Parameters that are to be cryptographically protected.
- **unprotected** (*dict*) – Parameters that are not cryptographically protected.
- **ciphertext** (*List[Any]*) – A cipher text.
- **sender_key** (*Optional[COSEKeyInterface]*) – A sender key as COSEKey.

Returns A recipient object.

Return type *RecipientInterface*

Raises **ValueError** – Invalid arguments.

classmethod from_jwk(*data: Union[str, bytes, Dict[str, Any]]*) → *cwt.recipient_interface.RecipientInterface*

Creates a recipient from JWK-like data.

Parameters **data** (*Union[str, bytes, Dict[str, Any]]*) – JSON-formatted recipient data.

Returns A recipient object.

Return type *RecipientInterface*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

classmethod from_list(*recipient: List[Any]*) → *cwt.recipient_interface.RecipientInterface*

Creates a recipient from a raw COSE array data.

Parameters **data** (*Union[str, bytes, Dict[str, Any]]*) – JSON-formatted recipient data.

Returns A recipient object.

Return type *RecipientInterface*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

class cwt.**Signer**(*cose_key:* cwt.cose_key_interface.COSEKeyInterface, *protected: Union[Dict[int, Any], bytes]*, *unprotected: Dict[int, Any]*, *signature: bytes = b''*)

Bases: cwt.cbor_processor.CBORProcessor

A Signer information.

property cose_key: ***cwt.cose_key_interface.COSEKeyInterface***

The COSE key for the signer.

property protected: **bytes**

The parameters that are to be cryptographically protected.

property unprotected: **Dict[int, Any]**

The parameters that are not cryptographically protected.

property signature: **bytes**

The signature that the signer signed.

classmethod new(*cose_key:* cwt.cose_key_interface.COSEKeyInterface, *protected: Union[dict, bytes] = {}*, *unprotected: dict = {}*, *signature: bytes = b''*)

Creates a signer information object (COSE_Signature).

Parameters

- **cose_key** (COSEKey) – A signature key for the signer.
- **protected** (*Union[dict, bytes]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*dict*) – Parameters that are not cryptographically protected.

- **signature** (*bytes*) – A signature as bytes.

Returns A signer information object.

Return type *Signer*

Raises **ValueError** – Invalid arguments.

classmethod from_jwk(*data: Union[str, bytes, Dict[str, Any]]*)

Creates a signer information object (COSE_Signature) from JWK. The `alg` in the JWK will be included in the protected header, and the `kid` in the JWT will be include in the unprotected header. If you want to include any other parameters in the protected/unprotected header, you have to use *Signer.new*.

Parameters **data** (*Union[str, bytes, Dict[str, Any]]*) – A JWK.

Returns A signer information object.

Return type *Signer*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

classmethod from_pem(*data: Union[str, bytes]*, *alg: Union[int, str] = ''*, *kid: Union[bytes, str] = b''*)

Creates a signer information object (COSE_Signature) from PEM-formatted key. The `alg` in the JWK will be included in the protected header, and the `kid` in the JWT will be include in the unprotected header. If you want to include any other parameters in the protected/unprotected header, you have to use *Signer.new*.

Parameters

- **data** (*Union[str, bytes]*) – A PEM-formatted key.
- **alg** (*Union[int, str]*) – An algorithm label(int) or name(str). It is only used when an algorithm cannot be specified by the PEM data, such as RSA family algorithms.
- **kid** (*Union[bytes, str]*) – A key identifier.

Returns A signer information object.

Return type *Signer*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode the key data.

sign(*msg: bytes*)

Returns a digital signature for the specified message using the specified key value.

Parameters **msg** (*bytes*) – A message to be signed.

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to sign the message.

verify(*msg: bytes*)

Verifies that the specified digital signature is valid for the specified message.

Parameters **msg** (*bytes*) – A message to be verified.

Raises

- **ValueError** – Invalid arguments.

- *VerifyError* – Failed to verify.

cwt.**load_pem_hcert_dsc**(*cert: Union[str, bytes]*) → *cwt.cose_key_interface.COSEKeyInterface*
Loads PEM-formatted DSC (Digital Signing Certificate) issued by CSCA (Certificate Signing Certificate Authority) as a COSEKey. At this time, the kid of the COSE key will be generated as a 8-byte truncated SHA256 fingerprint of the DSC complient with Electronic Health Certificate Specification.

Parameters **cert** (*str*) – A DSC.

Returns A DSC's public key as a COSE key.

Return type *COSEKeyInterface*

exception cwt.**CWTError**
Bases: Exception

Base class for all exceptions.

exception cwt.**EncodeError**
Bases: *cwt.exceptions.CWTError*

An Exception occurred when a CWT/COSE encoding process failed.

exception cwt.**DecodeError**
Bases: *cwt.exceptions.CWTError*

An Exception occurred when a CWT/COSE decoding process failed.

exception cwt.**VerifyError**
Bases: *cwt.exceptions.CWTError*

An Exception occurred when a verification process failed.

class cwt.cose_key_interface.**COSEKeyInterface**(*params: Dict[int, Any]*)
Bases: cwt.cbor_processor.CBORProcessor

The interface class for a COSE Key used for MAC, signing/verifying and encryption/decryption.

__init__(*params: Dict[int, Any]*)
Constructor.

Parameters **params** (*Dict[int, Any]*) – A COSE key common parameter object formatted to CBOR-like structure ((Dict[int, Any])).

property kty: int
The identifier of the key type.

property kid: Optional[bytes]
The key identifier.

property alg: Optional[int]
The algorithm that is used with the key.

property key_ops: List[int]
A set of permissible operations that the key is to be used for.

property base_iv: Optional[bytes]
Base IV to be xor-ed with Partial IVs.

property key: Any
The body of the key. It can be bytes or various PublicKey/PrivateKey objects defined in pyca/cryptography

to_dict() → Dict[int, Any]
Returns the CBOR-like structure (Dict[int, Any]) of the COSE key.

Returns The CBOR-like structure of the COSE key.

Return type Dict[int, Any]

`generate_nonce`() → bytes

Returns a nonce with the size suitable for the algorithm. This function will be called internally in *CWT* when no nonce is specified by the application. This function adopts `secrets.token_bytes()` to generate a nonce. If you do not want to use it, you should explicitly set a nonce to *CWT* functions (e.g., *encode_and_encrypt*).

Returns A byte string of the generated nonce.

Return type bytes

Raises **NotImplementedError** – Not implemented.

`sign`(*msg: bytes*) → bytes

Returns a digital signature for the specified message using the specified key value.

Parameters **msg** (*bytes*) – A message to be signed.

Returns The byte string of the encoded CWT.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- *EncodeError* – Failed to sign the message.

`verify`(*msg: bytes*, *sig: bytes*)

Verifies that the specified digital signature is valid for the specified message.

Parameters

- **msg** (*bytes*) – A message to be verified.
- **sig** (*bytes*) – A digital signature of the message.

Returns The byte string of the encoded CWT.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- *VerifyError* – Failed to verify.

`validate_certificate`(*ca_certs: List[bytes]*) → bool

Validate a certificate bound to the key with given trusted CA certificates if the key has *x5c* parameter.

Parameters **ca_certs** (*List[bytes]*) – A list of DER-formatted trusted root CA certificates which contains a concatenated list of trusted root certificates. You should specify private CA certificates in your target system. There should be no need to use the public CA certificates for the Web PKI.

Returns The indicator whether the validation is done or not.

Return type bool

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- *VerifyError* – Failed to verify.

encrypt(*msg: bytes*, *nonce: bytes*, *aad: bytes*) → bytes
Encrypts the specified message.

Parameters

- **msg** (*bytes*) – A message to be encrypted.
- **nonce** (*bytes*) – A nonce for encryption.
- **aad** (*bytes*) – Additional authenticated data.

Returns The byte string of encrypted data.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- *EncodeError* – Failed to encrypt the message.

decrypt(*msg: bytes*, *nonce: bytes*, *aad: bytes*) → bytes
Decrypts the specified message.

Parameters

- **msg** (*bytes*) – An encrypted message.
- **nonce** (*bytes*) – A nonce for encryption.
- **aad** (*bytes*) – Additional authenticated data.

Returns The byte string of the decrypted data.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- *DecodeError* – Failed to decrypt the message.

wrap_key(*key_to_wrap: bytes*) → bytes
Wraps a key.

Parameters **key_to_wrap** – A key to wrap.

Returns A wrapped key as bytes.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- *EncodeError* – Failed to derive key.

unwrap_key(*wrapped_key: bytes*) → bytes
Unwraps a key.

Parameters **wrapped_key** – A key to be unwrapped.

Returns An unwrapped key as bytes.

Return type bytes

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to unwrap key.

derive_key(*context: Union[List[Any], Dict[str, Any]]*, *material: bytes = b''*, *public_key: Optional[Any] = None*) → Any
Derives a key with a key material or key exchange.

Parameters

- **context** (*Union[List[Any], Dict[str, Any]]*) – Context information structure for key derivation functions.
- **material** (*bytes*) – A key material as bytes.
- **public_key** – A public key for key derivation with key exchange.

Returns A COSE key derived.

Return type *COSEKeyInterface*

Raises

- **NotImplementedError** – Not implemented.
- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to derive key.

class cwt.recipient_interface.**RecipientInterface**(*protected: Optional[Dict[int, Any]] = None*, *unprotected: Optional[Dict[int, Any]] = None*, *ciphertext: bytes = b''*, *recipients: List[Any] = []*, *key_ops: List[int] = []*, *key: bytes = b''*)

Bases: cwt.cbor_processor.CBORProcessor

The interface class for a COSE Recipient.

__init__(*protected: Optional[Dict[int, Any]] = None*, *unprotected: Optional[Dict[int, Any]] = None*, *ciphertext: bytes = b''*, *recipients: List[Any] = []*, *key_ops: List[int] = []*, *key: bytes = b''*)
Constructor.

Parameters

- **protected** (*Optional[Dict[int, Any]]*) – Parameters that are to be cryptographically protected.
- **unprotected** (*Optional[Dict[int, Any]]*) – Parameters that are not cryptographically protected.
- **ciphertext** – A ciphertext encoded as bytes.
- **recipients** – A list of recipient information structures.
- **key_ops** – A list of operations that the key is to be used for.

- **key** – A body of the key as bytes.

property kid: bytes
The key identifier.

property alg: int
The algorithm that is used with the key.

property protected: Dict[int, Any]
The parameters that are to be cryptographically protected.

property unprotected: Dict[int, Any]
The parameters that are not cryptographically protected.

property ciphertext: bytes
The ciphertext encoded as bytes

property recipients: Optional[List[Any]]
The list of recipient information structures.

to_list() → List[Any]
Returns the recipient information as a COSE recipient structure.

Returns The recipient structure.

Return type List[Any]

apply(*key: Optional[*cwt.cose_key_interface.COSEKeyInterface*] = None*, *recipient_key: Optional[*cwt.cose_key_interface.COSEKeyInterface*] = None*, *salt: Optional[bytes] = None*, *context: Optional[Union[List[Any], Dict[str, Any]]] = None*) → *cwt.cose_key_interface.COSEKeyInterface*
Applies a COSEKey as a material to prepare a MAC/encryption key with the recipient-specific method (e.g., key wrapping, key agreement, or the combination of them) and sets up the related information (context information or ciphertext) in the recipient structure. Therefore, it will be used by the sender of the recipient information before calling COSE.encode_* functions with the Recipient object. The key generated through this function will be set to **key** parameter of COSE.encode_* functions.

Parameters

- **key** (*Optional[*COSEKeyInterface*]*) – The external key to be used for preparing the key.
- **recipient_key** (*Optional[*COSEKeyInterface*]*) – The external public key provided by the recipient used for ECDH key agreement.
- **salt** (*Optional[bytes]*) – A salt used for deriving a key.
- **context** (*Optional[Union[List[Any], Dict[str, Any]]]*) – Context information structure.

Returns

A generated key or passed-throug key which is used as **key** parameter of COSE.encode_* functions.

Return type *COSEKeyInterface*

Raises

- **ValueError** – Invalid arguments.
- ***EncodeError*** – Failed to encode(e.g., wrap, derive) the key.

extract(*key:* cwt.cose_key_interface.COSEKeyInterface, *alg: Optional[int] = None*, *context: Optional[Union[List[Any], Dict[str, Any]]] = None*) → *cwt.cose_key_interface.COSEKeyInterface*

Extracts a MAC/encryption key with the recipient-specific method (e.g., key wrapping, key agreement, or the combination of them). This function will be called in COSE.decode so applications do not need to call it directly.

Parameters

- **key** (COSEKeyInterface) – The external key to be used for extracting the key.
- **alg** (*Optional[int]*) – The algorithm of the key extracted.
- **context** (*Optional[Union[List[Any], Dict[str, Any]]]*) – Context information structure.

Returns

An extracted key which is used for decrypting or verifying a payload message.

Return type *COSEKeyInterface*

Raises

- **ValueError** – Invalid arguments.
- ***DecodeError*** – Failed to decode(e.g., unwrap, derive) the key.

1.5 Supported CWT Claims

IANA Registry for CWT Claims lists all of registered CWT claims. This section shows the claims which this library currently supports. In particular, class `CWT` can validate the type of the claims and `Claims.from_json` can convert the following Names(str) into Values(int).

1.5.1 CBOR Web Token (CWT) Claims

Name	Status	Value	Description
hcert		-260	Health Certificate
EUPHNonce		-259	Challenge Nonce defined in FIDO Device Onboarding
EATMAROEPrefix		-258	Signing prefix for multi-app restricted operating environments
EAT-FDO		-257	EAT-FDO may contain related to FIDO Device Onboarding
iss		1	Issuer
sub		2	Subject
aud		3	Audience
exp		4	Expiration Time
nbf		5	Not Before
iat		6	Issued At
cti		7	CWT ID
cnf		8	Confirmation
nonce		10	Nonce
ueid		11	Universal Entity ID Claim
oemid		13	OEM Identification by IEEE
seclevel		14	Security Level
secboot		15	Secure Boot
dbgstat		16	Debug Status
location		17	Location
eat_profile		18	EAT Profile
submods		20	The Submodules Part of a Token

1.5.2 CWT Confirmation Methods

Name	Status	Value	Description
COSE_Key		1	COSE_Key Representing Public Key
Encrypted_COSE_Key		2	Encrypted COSE_Key
kid		3	Key Identifier

1.6 Supported COSE Algorithms

IANA Registry for COSE lists many cryptographic algorithms for MAC, signing, and encryption. This section shows the algorithms which this library currently supports.

- : Supported.
- : No plan to support.

1.6.1 COSE Key Types

Name	Status	Value	Description
OKP		1	Octet Key Pair
EC2		2	Elliptic Curve Keys w/ x- and y-coordinate pair
RSA		3	RSA Key
Symmetric		4	Symmetric Keys
HSS-LMS		5	Public key for HSS/LMS hash-based digital signature
WalnutDSA		6	WalnutDSA public key

1.6.2 COSE Algorithms

Name	Status	Value	Description
RS1		-65535	RSASSA-PKCS1-v1_5 using SHA-1
WalnutDSA		-260	WalnutDSA signature
RS512		-259	RSASSA-PKCS1-v1_5 using SHA-512
RS384		-258	RSASSA-PKCS1-v1_5 using SHA-384
RS256		-257	RSASSA-PKCS1-v1_5 using SHA-256
ES256K		-47	ECDSA using secp256k1 curve and SHA-256
HSS-LMS		-46	HSS/LMS hash-based digital signature
SHAKE256		-45	SHAKE-256 512-bit Hash Value
SHA-512		-44	SHA-2 512-bit Hash
SHA-384		-43	SHA-2 384-bit Hash
RSAES-OAEP w/ SHA-512		-42	RSAES-OAEP w/ SHA-512
RSAES-OAEP w/ SHA-256		-41	RSAES-OAEP w/ SHA-256
RSAES-OAEP w/ RFC 8017 default parameters		-40	RSAES-OAEP w/ SHA-1

continues on next page

Table 1 – continued from previous page

Name	Status	Value	Description
PS512		-39	RSASSA-PSS w/ SHA-512
PS384		-38	RSASSA-PSS w/ SHA-384
PS256		-37	RSASSA-PSS w/ SHA-256
ES512		-36	ECDSA w/ SHA-512
ES384		-35	ECDSA w/ SHA-384
ECDH-SS + A256KW		-34	ECDH SS w/ Concat KDF and AES Key Wrap w/ 256-bit key
ECDH-SS + A192KW		-33	ECDH SS w/ Concat KDF and AES Key Wrap w/ 192-bit key
ECDH-SS + A128KW		-32	ECDH SS w/ Concat KDF and AES Key Wrap w/ 128-bit key
ECDH-ES + A256KW		-31	ECDH ES w/ Concat KDF and AES Key Wrap w/ 256-bit key
ECDH-ES + A192KW		-30	ECDH ES w/ Concat KDF and AES Key Wrap w/ 192-bit key
ECDH-ES + A128KW		-29	ECDH ES w/ Concat KDF and AES Key Wrap w/ 128-bit key
ECDH-SS + HKDF-512		-28	ECDH SS w/ HKDF - generate key directly
ECDH-SS + HKDF-256		-27	ECDH SS w/ HKDF - generate key directly
ECDH-ES + HKDF-512		-26	ECDH ES w/ HKDF - generate key directly
ECDH-ES + HKDF-256		-25	ECDH ES w/ HKDF - generate key directly
SHAKE128		-18	SHAKE-128 256-bit Hash Value

continues on next page

Table 1 – continued from previous page

Name	Status	Value	Description
SHA-512/256		-17	SHA-2 512-bit Hash truncated to 256-bits
SHA-256		-16	SHA-2 256-bit Hash
SHA-256/64		-15	SHA-2 256-bit Hash truncated to 64-bits
SHA-1		-14	SHA-1 Hash
direct+HKDF-AES-256		-13	Shared secret w/ AES-MAC 256-bit key
direct+HKDF-AES-128		-12	Shared secret w/ AES-MAC 128-bit key
direct+HKDF-SHA-512		-11	Shared secret w/ HKDF and SHA-512
direct+HKDF-SHA-256		-10	Shared secret w/ HKDF and SHA-256
EdDSA		-8	EdDSA
ES256		-7	ECDSA w/ SHA-256
direct		-6	Direct use of CEK
A256KW		-5	AES Key Wrap w/ 256-bit key
A192KW		-4	AES Key Wrap w/ 192-bit key
A128KW		-3	AES Key Wrap w/ 128-bit key
A128GCM		1	AES-GCM mode w/ 128-bit key, 128-bit tag
A192GCM		2	AES-GCM mode w/ 192-bit key, 128-bit tag
A256GCM		3	AES-GCM mode w/ 256-bit key, 128-bit tag
HMAC 256/64		4	HMAC w/ SHA-256 truncated to 64 bits
HMAC 256/256 (“HS256” can also be used.)		5	HMAC w/ SHA-256
HMAC 384/384 (“HS384” can also be used.)		6	HMAC w/ SHA-384
HMAC 512/512 (“HS512” can also be used.)		7	HMAC w/ SHA-512

continues on next page

Table 1 – continued from previous page

Name	Status	Value	Description
AES-CCM-16-64-128		10	AES-CCM mode 128-bit key, 64-bit tag, 13-byte nonce
AES-CCM-16-64-256		11	AES-CCM mode 256-bit key, 64-bit tag, 13-byte nonce
AES-CCM-64-64-128		12	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
AES-CCM-64-64-256		13	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce
AES-MAC 128/64		14	AES-MAC 128-bit key, 64-bit tag
AES-MAC 256/64		15	AES-MAC 256-bit key, 64-bit tag
ChaCha20/Poly1305		24	ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag
AES-MAC 128/128		25	AES-MAC 128-bit key, 128-bit tag
AES-MAC 128/128		26	AES-MAC 256-bit key, 128-bit tag
AES-CCM-16-128-128		30	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
AES-CCM-16-128-256		31	AES-CCM mode 256-bit key, 128-bit tag, 13-byte nonce
AES-CCM-64-128-128		32	AES-CCM mode 128-bit key, 128-bit tag, 7-byte nonce

continues on next page

Table 1 – continued from previous page

Name	Status	Value	Description
AES-CCM-64-128-256		33	AES-CCM mode 256-bit key, 128-bit tag, 7-byte nonce

1.6.3 COSE Elliptic Curves

Name	Status	Value	Description
P-256		1	NIST P-256 also known as secp256r1
P-384		2	NIST P-384 also known as secp384r1
P-521		3	NIST P-521 also known as secp521r1
X25519		4	X25519 for use w/ ECDH only
X448		5	X448 for use w/ ECDH only
Ed25519		6	Ed25519 for use w/ EdDSA only
Ed448		7	Ed448 for use w/ EdDSA only
secp256k1		8	SECG secp256k1 curve

1.7 Referenced Specifications

This library is (partially) compliant with following specifications:

- RFC8152: CBOR Object Signing and Encryption (COSE)
- RFC8230: Using RSA Algorithms with COSE Messages
- RFC8392: CBOR Web Token (CWT)
- RFC8747: Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)
- RFC8812: COSE and JOSE Registrations for Web Authentication (WebAuthn) Algorithms

1.8 Changes

1.8.1 Unreleased

1.8.2 Version 1.4.2

Released 2021-10-16

- Add support for Python 3.10. #183

1.8.3 Version 1.4.1

Released 2021-10-11

- Make public types explicit for PyLance. #180
- Use datetime.now(tz=timezone.utc) instead of datetime.utcnow. #179
- Add py.typed for PEP561. #176

1.8.4 Version 1.4.0

Released 2021-10-04

- Add support for x5c. #174

1.8.5 Version 1.3.2

Released 2021-08-09

- Add support for byte-formatted kid on from_jwk(). #165
- Add sample of EUDCC verifier. #160

1.8.6 Version 1.3.1

Released 2021-07-07

- Fix docstring for CWT, COSE, etc. #158
- Add PS256 support for hcert. #156

1.8.7 Version 1.3.0

Released 2021-07-03

- Add helper for hcert. #154

1.8.8 Version 1.2.0

Released 2021-07-01

- Disable access to CWT property for global CWT instance (cwt). #153
- Fix kid verification for recipient. #152
- Change default setting of verify_kid to True for CWT. #150
- Add setter/getter for each setting to COSE/CWT. #150
- Fix type of parameter for COSE constructor. #149
- Add verify_kid option to COSE. #148
- Fix kid verification. #148
- Add support for hcert. #147

1.8.9 Version 1.1.0

Released 2021-06-27

- Add context support to Recipient.from_jwk(). #144
- Disable auto salt generation in the case of ECDH-ES. #143
- Add support for auto salt generation. #142
- Add salt parameter to RecipientInterface.apply(). #142
- Remove alg parameter from RecipientInterface.apply(). #141

1.8.10 Version 1.0.0

Released 2021-06-24

- Make MAC key can be derived with ECDH. #139
- Add RawKey for key material. #138
- Make MAC key can be derived with HKDF. #137
- Remove COSEKeyInterface from RecipientInterface. #137
- Implement AESKeyWrap which has COSEKeyInterface. #137
- Add encode_key() to RecipientInterface. #134
- Rename key to keys on CWT/COSE decode(). #133
- Remove materials from COSE.decode(). #131
- Add decode_key() to RecipientInterface. #131
- Remove alg from keys in recipient header. #131
- Add support for ECDH with key wrap. #130
- Refine README. #127
- Add samples of using direct key agreement. #126

1.8.11 Version 0.10.0

Released 2021-06-13

- Rename from_json to from_jwk. #124
- Add support for X25519/X448. #123
- Add derive_key to EC2Key. #122
- Add key to OKPKey. #122
- Add support for key derivation without kid. #120
- Add support for ECDH-SS direct HKDF. #119
- Add support for ECDH-ES direct HKDF. #118

1.8.12 Version 0.9.0

Released 2021-06-04

- Introduce new() into CWT/COSE. #115
- Rename Claims.from_dict to Claims.new. #115
- Rename COSEKey.from_dict to COSEKey.new. #115
- Rename Recipient.from_dict to Recipient.new. #115
- Add Signer for encode_and_sign function. #114
- Divide CWT options into independent parameters. #113

1.8.13 Version 0.8.1

Released 2021-05-31

- Add JSON support for COSE. #109
- Devite a COSE options parameter into independent parameters. #109
- Refine COSE default mode. #108
- Refine the order of parameters for CWT functions. #107
- Fix example in docstring. #107
- Make interface docstring public. #106

1.8.14 Version 0.8.0

Released 2021-05-30

- Refine EncryptedCOSEKey interface. #104
- Merge RecipientsBuilder into Recipients. #103
- Rename Key to COSEKeyInterface. #102
- Rename RecipientBuilder to Recipient. #101
- Make Key private. #100
- Merge ClaimsBuilder into Claims. #98
- Rename KeyBuilder to COSEKey. #97
- Rename COSEKey to Key. #97
- Add support for external AAD. #94
- Make unwrap_key return COSEKey. #93
- Fix default HMAC key size. #91
- Add support for AES key wrap. #89
- Add support for direct+HKDF-SHA256 and SHA512. #87

1.8.15 Version 0.7.1

Released 2021-05-11

- Add alg validation and fix related bug. #77
- Update protected/unprotected default value from {} to None. #76

1.8.16 Version 0.7.0

Released 2021-05-09

- Add support for bytes-formatted protected header. #73
- Derive alg from kty and crv on from_jwk. #73
- Add alg_auto_inclusion. #73
- Move nonce generation from CWT to COSE. #73
- Re-order arguments of COSE API. #73
- Add support for COSE algorithm names for KeyBuilder.from_jwk. #72
- Add tests based on COSE WG examples. #72
- Move parameter auto-gen function from CWT to COSE. #72
- Refine COSE API to make the type of payload parameter be bytes only. #71
- Simplify samples on docs. #69

1.8.17 Version 0.6.1

Released 2021-05-08

- Add test for error handling of encoding/decoding. #67
- Fix low level error message. #67
- Add support for multiple aud. #65
- Relax the condition of the acceptable private claim value. #64
- Fix doc version. #63

1.8.18 Version 0.6.0

Released 2021-05-04

- Make decode accept multiple keys. #61
- Add set_private_claim_names to ClaimsBuilder and CWT. #60
- Add sample of CWT with user-defined claims to docs. #60

1.8.19 Version 0.5.0

Released 2021-05-04

- Make ClaimsBuilder return Claims. #56
- Add support for JWK keyword of alg and key_ops. #55
- Add from_jwk. #53
- Add support for PoP key (cnf claim). #50
- Add to_dict to COSEKey. #50
- Add crv property to COSEKey. #50
- Add key property to COSEKey. #50
- Add support for RSASSA-PSS. #49
- Add support for RSASSA-PKCS1-v1_5. #48

1.8.20 Version 0.4.0

Released 2021-04-30

- Add CWT.encode. #46
- Fix bug on KeyBuilder.from_dict. #45
- Add support for key_ops. #44
- Add support for ChaCha20/Poly1305. #43
- Make nonce optional for CWT.encode_and_encrypt. #42
- Add support for AES-GCM (A128GCM, A192GCM and A256GCM). #41
- Make key optional for KeyBuilder.from_symmetric_key. #41

1.8.21 Version 0.3.0

Released 2021-04-29

- Add docstring to COSE, KeyBuilder and more. #39
- Add support for COSE_Encrypt structure. #36
- Add support for COSE_Signature structure. #35
- Change protected_header type from bytes to dict. #34
- Add support for COSE_Mac structure. #32
- Add test for CWT. #29

1.8.22 Version 0.2.3

Released 2021-04-23

- Add test for cose_key and fix bugs. #21
- Add support for exp, nbf and iat. #18

1.8.23 Version 0.2.2

Released 2021-04-19

- Add support for Ed448, ES384 and ES512. #13
- Add support for EncodeError and DecodeError. #13
- Add test for supported algorithms. #13
- Update supported algorithms and claims on docs. #13

1.8.24 Version 0.2.1

Released 2021-04-18

- Add VerifyError. #11
- Fix HMAC alg names. #11
- Make COSEKey public. #11
- Add tests for HMAC. #11

1.8.25 Version 0.2.0

Released 2021-04-18

- Add docs for CWT. #9
- Raname exceptions. #9

1.8.26 Version 0.1.1

Released 2021-04-18

- Fix description of installation.

1.8.27 Version 0.1.0

Released 2021-04-18

- First public preview release.

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F