
Swift-Sodium build passing

Swift-Sodium provides a safe and easy to use interface to perform common cryptographic operations on macOS, iOS, tvOS and watchOS.

It leverages the Sodium library, and although Swift is the primary target, the framework can also be used in Objective-C applications.

Please help!

The current Swift-Sodium documentation is not great. Your help to improve it and make it awesome would be very appreciated!

Usage

To add Swift-Sodium as dependency to your Xcode project, select **File > Swift Packages > Add Package Dependency**, enter its repository URL: <https://github.com/jedisct1/swift-sodium.git> and import **Sodium** as well as **Clibsodium**.

Then, to use it in your source code, add:

```
1 import Sodium
```

The Sodium library itself doesn't have to be installed on the system: the repository already includes a precompiled library for armv7, armv7s, arm64, as well as for the iOS simulator, WatchOS and Catalyst.

The **Clibsodium.xcframework** framework has been generated by the `dist-build/apple-xcframework.sh` script.

Running this script on Xcode 15.2 on the revision [b3333f07fae1daf1b46ec1ee51ddff0733e73832](#) of libsodium generates files identical to the ones present in this repository.

Secret-key cryptography

Messages are encrypted and decrypted using the same secret key, this is also known as symmetric cryptography.

A key can be generated using the `key()` method, derived from a password using the Password Hashing API, or computed using a secret key and the peer's public key utilising the Key Exchange API.

Authenticated encryption for a sequence of messages

```
1 let sodium = Sodium()
2 let message1 = "Message 1".bytes
3 let message2 = "Message 2".bytes
4 let message3 = "Message 3".bytes
5
6 let secretkey = sodium.secretStream.xchacha20poly1305.key()
7
8 /* stream encryption */
9
10 let stream_enc = sodium.secretStream.xchacha20poly1305.initPush(
    secretKey: secretkey)!
11 let header = stream_enc.header()
12 let encrypted1 = stream_enc.push(message: message1)!
13 let encrypted2 = stream_enc.push(message: message2)!
14 let encrypted3 = stream_enc.push(message: message3, tag: .FINAL)!
15
16 /* stream decryption */
17
18 let stream_dec = sodium.secretStream.xchacha20poly1305.initPull(
    secretKey: secretkey, header: header)!
19 let (message1_dec, tag1) = stream_dec.pull(cipherText: encrypted1)!
20 let (message2_dec, tag2) = stream_dec.pull(cipherText: encrypted2)!
21 let (message3_dec, tag3) = stream_dec.pull(cipherText: encrypted3)!
```

A stream is a sequence of messages, that will be encrypted as they depart, and, decrypted as they arrive. The encrypted messages are expected to be received in the same order as they were sent.

Streams can be arbitrarily long. This API can thus be used for file encryption, by splitting files into small chunks, so that the whole file doesn't need to reside in memory concurrently.

It can also be used to exchange a sequence of messages between two peers.

The decryption function automatically checks that chunks have been received without modification, and truncation or reordering.

A tag is attached to each message, and can be used to signal the end of a sub-sequence ([PUSH](#)), or the end of the string ([FINAL](#)).

Authenticated encryption for single messages

```
1 let sodium = Sodium()
2 let message = "My Test Message".bytes
3 let secretKey = sodium.secretBox.key()
4 let encrypted: Bytes = sodium.secretBox.seal(message: message,
    secretKey: secretKey)!
```

```
5 if let decrypted = sodium.secretBox.open(
    nonceAndAuthenticatedCipherText: encrypted, secretKey: secretKey) {
6     // authenticator is valid, decrypted contains the original message
7 }
```

This API encrypts a message. The decryption process will check that the messages haven't been tampered with before decrypting them.

Messages encrypted this way are independent: if multiple messages are sent this way, the recipient cannot detect if some messages have been duplicated, deleted or reordered without the sender including additional data with each message.

Optionally, `SecretBox` provides the ability to utilize a user-defined nonce via `seal(message: secretKey: nonce:)`.

Public-key Cryptography

With public-key cryptography, each peer has two keys: a secret (private) key, that has to remain secret, and a public key that anyone can use to send an encrypted message to that peer. That public key can be only be used to encrypt a message. The corresponding secret is required to decrypt it.

Authenticated Encryption

```
1 let sodium = Sodium()
2 let aliceKeyPair = sodium.box.keyPair()!
3 let bobKeyPair = sodium.box.keyPair()!
4 let message = "My Test Message".bytes
5
6 let encryptedMessageFromAliceToBob: Bytes =
7     sodium.box.seal(message: message,
8                     recipientPublicKey: bobKeyPair.publicKey,
9                     senderSecretKey: aliceKeyPair.secretKey)!
10
11 let messageVerifiedAndDecryptedByBob =
12     sodium.box.open(nonceAndAuthenticatedCipherText:
13                     encryptedMessageFromAliceToBob,
14                     senderPublicKey: aliceKeyPair.publicKey,
15                     recipientSecretKey: bobKeyPair.secretKey)
```

This operation encrypts and sends a message to someone using their public key.

The recipient has to know the sender's public key as well, and will reject a message that doesn't appear to be valid for the expected public key.

`seal()` automatically generates a nonce and prepends it to the ciphertext. `open()` extracts the nonce and decrypts the ciphertext.

Optionally, `Box` provides the ability to utilize a user-defined nonce via `seal(message: recipientPublicKey: senderSecretKey: nonce:)`.

The `Box` class also provides alternative functions and parameters to deterministically generate key pairs, to retrieve the nonce and/or the authenticator, and to detach them from the original message.

Anonymous Encryption (Sealed Boxes)

```
1 let sodium = Sodium()
2 let bobKeyPair = sodium.box.keyPair()!
3 let message = "My Test Message".bytes
4
5 let encryptedMessageToBob =
6     sodium.box.seal(message: message, recipientPublicKey: bobKeyPair.
7         publicKey)!
8
9 let messageDecryptedByBob =
10     sodium.box.open(anonymousCipherText: encryptedMessageToBob,
11         recipientPublicKey: bobKeyPair.publicKey,
12         recipientSecretKey: bobKeyPair.secretKey)
```

`seal()` generates an ephemeral keypair, uses the ephemeral secret key in the encryption process, combines the ephemeral public key with the ciphertext, then destroys the keypair.

The sender cannot decrypt the resulting ciphertext. `open()` extracts the public key and decrypts using the recipient's secret key. Message integrity is verified, but the sender's identity cannot be correlated to the ciphertext.

Key exchange

```
1 let sodium = Sodium()
2 let aliceKeyPair = sodium.keyExchange.keyPair()!
3 let bobKeyPair = sodium.keyExchange.keyPair()!
4
5 let sessionKeyPairForAlice = sodium.keyExchange.sessionKeyPair(
6     publicKey: aliceKeyPair.publicKey,
7     secretKey: aliceKeyPair.secretKey, otherPublicKey: bobKeyPair.
8         publicKey, side: .CLIENT)!
9
10 let sessionKeyPairForBob = sodium.keyExchange.sessionKeyPair(publicKey:
11     bobKeyPair.publicKey,
12     secretKey: bobKeyPair.secretKey, otherPublicKey: aliceKeyPair.
13         publicKey, side: .SERVER)!
14
15
```

```
10 let aliceToBobKeyEquality = sodium.utils.equals(sessionKeyPairForAlice.  
    tx, sessionKeyPairForBob.rx) // true  
11 let bobToAliceKeyEquality = sodium.utils.equals(sessionKeyPairForAlice.  
    rx, sessionKeyPairForBob.tx) // true
```

Public-key signatures

Signatures allow multiple parties to verify the authenticity of a public message, using the public key of the author's message.

This can be especially useful to sign software updates.

Detached signatures

The signature is generated separately to the original message.

```
1 let sodium = Sodium()  
2 let message = "My Test Message".bytes  
3 let keyPair = sodium.sign.keyPair()!  
4 let signature = sodium.sign.signature(message: message, secretKey:  
    keyPair.secretKey)!  
5 if sodium.sign.verify(message: message,  
6                        publicKey: keyPair.publicKey,  
7                        signature: signature) {  
8     // signature is valid  
9 }
```

Attached signatures

The signature is generated and prepended to the original message.

```
1 let sodium = Sodium()  
2 let message = "My Test Message".bytes  
3 let keyPair = sodium.sign.keyPair()!  
4 let signedMessage = sodium.sign.sign(message: message, secretKey:  
    keyPair.secretKey)!  
5 if let unsignedMessage = sodium.sign.open(signedMessage: signedMessage,  
6      publicKey: keyPair.publicKey) {  
6     // signature is valid  
7 }
```

Hashing

Deterministic hashing

Hashing effectively “fingerprints” input data, no matter what its size, and returns a fixed length “digest”.

The digest length can be configured as required, from 16 to 64 bytes.

```
1 let sodium = Sodium()
2 let message = "My Test Message".bytes
3 let hash = sodium.genericHash.hash(message: message)
4 let hashOfSize32Bytes = sodium.genericHash.hash(message: message,
    outputLength: 32)
```

Keyed hashing

```
1 let sodium = Sodium()
2 let message = "My Test Message".bytes
3 let key = "Secret key".bytes
4 let h = sodium.genericHash.hash(message: message, key: key)
```

Streaming

```
1 let sodium = Sodium()
2 let message1 = "My Test ".bytes
3 let message2 = "Message".bytes
4 let key = "Secret key".bytes
5 let stream = sodium.genericHash.initStream(key: key)!
6 stream.update(input: message1)
7 stream.update(input: message2)
8 let h = stream.final()
```

Short-output hashing (SipHash)

```
1 let sodium = Sodium()
2 let message = "My Test Message".bytes
3 let key = sodium.randomBytes.buf(length: sodium.shortHash.KeyBytes)!
4 let h = sodium.shortHash.hash(message: message, key: key)
```

Random numbers generation

Random number generation produces cryptographically secure pseudorandom numbers suitable as key material.

```
1 let sodium = Sodium()
2 let randomBytes = sodium.randomBytes.buf(length: 1000)!
3 let seed = "0123456789abcdef0123456789abcdef".bytes
4 let stream = sodium.randomBytes.deterministic(length: 1000, seed: seed)
  !
```

Use `RandomBytes.Generator` as a generator to produce cryptographically secure pseudorandom numbers.

```
1 var rng = RandomBytes.Generator()
2 let randomUInt32 = UInt32.random(in: 0...10, using: &rng)
3 let randomUInt64 = UInt64.random(in: 0...10, using: &rng)
4 let randomInt = Int.random(in: 0...10, using: &rng)
5 let randomDouble = Double.random(in: 0...1, using: &rng)
```

Password hashing

Password hashing provides the ability to derive key material from a low-entropy password. Password hashing functions are designed to be expensive to hamper brute force attacks, thus the computational and memory parameters may be user-defined.

```
1 let sodium = Sodium()
2 let password = "Correct Horse Battery Staple".bytes
3 let hashedStr = sodium.pwHash.str(passwd: password,
4                                   opsLimit: sodium.pwHash.
5                                       OpsLimitInteractive,
6                                   memLimit: sodium.pwHash.
7                                       MemLimitInteractive)!
8
9 if sodium.pwHash.strVerify(hash: hashedStr, passwd: password) {
10     // Password matches the given hash string
11 } else {
12     // Password doesn't match the given hash string
13 }
14
15 if sodium.pwHash.strNeedsRehash(hash: hashedStr,
16                                 opsLimit: sodium.pwHash.
17                                     OpsLimitInteractive,
18                                 memLimit: sodium.pwHash.
19                                     MemLimitInteractive) {
20     // Previously hashed password should be recomputed because the way
21     it was
```

```
17 // hashed doesn't match the current algorithm and the given
18 } parameters.
```

Authentication tags

The `sodium.auth.tag()` function computes an authentication tag (HMAC) using a message and a key. Parties knowing the key can then verify the authenticity of the message using the same parameters and the `sodium.auth.verify()` function.

Authentication tags are not signatures: the same key is used both for computing and verifying a tag. Therefore, verifiers can also compute tags for arbitrary messages.

```
1 let sodium = Sodium()
2 let input = "test".bytes
3 let key = sodium.auth.key()
4 let tag = sodium.auth.tag(message: input, secretKey: key)!
5 let tagIsValid = sodium.auth.verify(message: input, secretKey: key, tag
: tag)
```

Key derivation

The `sodium.keyDerivation.derive()` function generates a subkey using an input (master) key, an index, and a 8 bytes string identifying the context. Up to $(2^{64}) - 1$ subkeys can be generated for each context, by incrementing the index.

```
1 let sodium = Sodium()
2 let secretKey = sodium.keyDerivation.keygen()!
3
4 let subKey1 = sodium.keyDerivation.derive(secretKey: secretKey,
5                                           index: 0, length: 32,
6                                           context: "Context!")
7 let subKey2 = sodium.keyDerivation.derive(secretKey: secretKey,
8                                           index: 1, length: 32,
9                                           context: "Context!")
```

Utilities

Zeroing memory

```
1 let sodium = Sodium()
2 var dataToZero = "Message".bytes
3 sodium.utils.zero(&dataToZero)
```

Constant-time comparison

```
1 let sodium = Sodium()
2 let secret1 = "Secret key".bytes
3 let secret2 = "Secret key".bytes
4 let equality = sodium.utils.equals(secret1, secret2)
```

Padding

```
1 let sodium = Sodium()
2 var bytes = "test".bytes
3
4 // make bytes.count a multiple of 16
5 sodium.utils.pad(bytes: &bytes, blockSize: 16)!
6
7 // restore original size
8 sodium.utils.unpad(bytes: &bytes, blockSize: 16)!
```

Padding can be useful to hide the length of a message before it is encrypted.

Constant-time hexadecimal encoding

```
1 let sodium = Sodium()
2 let bytes = "Secret key".bytes
3 let hex = sodium.utils.bin2hex(bytes)
```

Hexadecimal decoding

```
1 let sodium = Sodium()
2 let data1 = sodium.utils.hex2bin("deadbeef")
3 let data2 = sodium.utils.hex2bin("de:ad be:ef", ignore: " :")
```

Constant-time base64 encoding

```
1 let sodium = Sodium()
2 let b64 = sodium.utils.bin2base64("data".bytes)!
3 let b64_2 = sodium.utils.bin2base64("data".bytes, variant: .
    URLSAFE_NO_PADDING)!
```

Base64 decoding

```
1 let data1 = sodium.utils.base642bin(b64)
2 let data2 = sodium.utils.base642bin(b64, ignore: " \n")
3 let data3 = sodium.utils.base642bin(b64_2, variant: .URLSAFE_NO_PADDING
  , ignore: " \n")
```

Helpers to build custom constructions

Only use the functions below if you know that you absolutely need them, and know how to use them correctly.

Unauthenticated encryption

The `sodium.stream.xor()` function combines (using the XOR operation) an arbitrary-long input with the output of a deterministic key stream derived from a key and a nonce. The same operation applied twice produces the original input.

No authentication tag is added to the output. The data can be tampered with; an adversary can flip arbitrary bits.

In order to encrypt data using a secret key, the `SecretBox` class is likely to be what you are looking for.

In order to generate a deterministic stream out of a seed, the `RandomBytes.deterministic_rand()` function is likely to be what you need.

```
1 let sodium = Sodium()
2 let input = "test".bytes
3 let key = sodium.stream.key()
4 let (output, nonce) = sodium.stream.xor(input: input, secretKey: key)!
5 let twice = sodium.stream.xor(input: output, nonce: nonce, secretKey:
  key)!
6
7 XCTAssertEqual(input, twice)
```

Algorithms

- Stream ciphers: XChaCha20, XSalsa20
- AEADs: XChaCha20Poly1305, AEGIS-128L, AEGIS-256, AES256-GCM
- MACs: Poly1305, HMAC-SHA512/256
- Hash function: BLAKE2B

-
- Key exchange: X25519
 - Signatures: Ed25519