

p-pqc

Multi-platform library integration guide

Swift / iOS | Kotlin / Android | Flutter | WASM / Browser | Go

Algorithm	ML-KEM-768 (FIPS 203) + AES-256-GCM
Standard	NIST FIPS 203 (ML-KEM) — Level 3
Core language	Rust — Pure Rust, no native libs
Platforms	iOS · Android · Flutter · WASM · Go · Windows · macOS
Contact	letanphuoc@gmail.com

Version 02/11/2026

p-pqc — Library Integration Guide

Introduction

p-pqc is a Post-Quantum Cryptography library that is ready to be embedded into your application. Developers only need to grab the pre-built library file and call the API — no need to install Rust or compile anything.

Algorithm	Description
CRYSTALS-Kyber-768	Key Encapsulation Mechanism — NIST FIPS 203 Level 3
AES-256-GCM	Authenticated Encryption (AEAD) — NIST SP 800-38D
Combined	ML-KEM generates the shared secret, AES-256-GCM encrypts the data; randomness from HMAC_DRBG SHA-512 (SP 800-90A)

Security properties:

- Resistant to quantum-computer attacks (Grover, Shor algorithm)
- Integrity authentication — detects data tampering (16-byte AEAD tag)
- Forward secrecy — each encryption creates a completely new shared secret
- Cross-platform compatibility — ciphertext produced by Swift can be read by Kotlin, Go, Flutter

---

Library File List

You will receive the following set of files from the backend / release team:

Platform	Files needed	Notes
Windows x64	`p_pqc.dll` , `p_pqc.lib` , `p_pqc.h`	Dynamic library
macOS	`libp_pqc.a` , `p_pqc.h`	Static — universal arm64 + x86_64
macOS (dynamic)	`libp_pqc.dylib` , `p_pqc.h`	Dynamic library
iOS	`PPQC.xcframework/` , `p_pqc.h`	XCFramework — device + simulator
Android	`jniLibs/arm64-v8a/libp_pqc.so`	Physical 64-bit devices
Android	`jniLibs/armeabi-v7a/libp_pqc.so`	Physical 32-bit devices
Android	`jniLibs/x86_64/libp_pqc.so`	64-bit emulator

Platform	Files needed	Notes
Android	`jniLibs/x86/libp_pqc.so`	32-bit emulator
Flutter Android	`jniLibs/*/libp_pqc.so`	C ABI — dart:ffi, no JNI
Flutter iOS	`PPQC.xcframework/`	Same as iOS native
Go	`libp_pqc.a` , `p_pqc.h`	CGo static linking

Header file `p_pqc.h` — API summary:

```
// Key sizes (bytes)
#define PPQC_PK_LEN  1184   // Public key
#define PPQC_SK_LEN  2400   // Secret key
#define PPQC_CT_LEN  1088   // KEM ciphertext
#define PPQC_SS_LEN  32     // Shared secret

// Generate a key pair — returns 0=OK, -1=error
int ppqc_keygen(uint8_t *pk_out, uint8_t *sk_out);

// Compute the output buffer size before calling ppqc_encrypt
size_t ppqc_encrypt_output_len(size_t plaintext_len);

// Encrypt — returns 0=OK, -1=error
int ppqc_encrypt(const uint8_t *pk,
                 const uint8_t *plaintext, size_t plaintext_len,
                 uint8_t *out, size_t *out_len);

// Decrypt — returns 0=OK, -1=error (wrong key or tampered data)
int ppqc_decrypt(const uint8_t *sk,
                 const uint8_t *ciphertext, size_t ciphertext_len,
                 uint8_t *out, size_t *out_len);

// Raw KEM (optional — manage symmetric encryption yourself)
int ppqc_kem_encapsulate(const uint8_t *pk,
                         uint8_t *ct_out, uint8_t *ss_out);
int ppqc_kem_decapsulate(const uint8_t *sk,
                         const uint8_t *ct, uint8_t *ss_out);
```

Windows — C / C++

Setup

```
MyProject/
├── main.c
├── p_pqc.h       ← copy here
├── p_pqc.lib     ← import library for MSVC
└── p_pqc.dll     ← copy into the exe's run directory
```

Visual Studio

Project Properties → Linker → Input → Additional Dependencies:

```
p_pqc.lib
```

Project Properties → C/C++ → General → Additional Include Directories:

```
$(ProjectDir)
```

Sample C code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "p_pqc.h"

int main(void) {
    // 1. Generate a key pair
    uint8_t pk[PPQC_PK_LEN];
    uint8_t sk[PPQC_SK_LEN];
    if (ppqc_keygen(pk, sk) != 0) {
        fprintf(stderr, "Keygen failed\n");
        return 1;
    }
    printf("Public key: %d bytes\n", PPQC_PK_LEN);

    // 2. Encrypt
    const char *message = "Hello p-pqc!";
    size_t pt_len  = strlen(message);
    size_t buf_len = ppqc_encrypt_output_len(pt_len);
    uint8_t *blob  = malloc(buf_len);

    size_t out_len = buf_len;
    if (ppqc_encrypt(pk, (const uint8_t *)message, pt_len, blob, &out_len) != 0) {
        fprintf(stderr, "Encryption failed\n");
        free(blob); return 1;
    }
    printf("Ciphertext: %zu bytes\n", out_len);

    // 3. Decrypt
    uint8_t *plain = malloc(out_len);
    size_t   plain_len = out_len;
    if (ppqc_decrypt(sk, blob, out_len, plain, &plain_len) != 0) {
        fprintf(stderr, "Decryption failed\n");
        free(blob); free(plain); return 1;
    }
    printf("Decrypted: %.*s\n", (int)plain_len, plain);

    free(blob); free(plain);
    return 0;
}
```

Windows — .NET / C# (P/Invoke)

```
using System;
using System.Runtime.InteropServices;
using System.Text;

public static class PPQC
{
    private const string Dll = "p_pqc";

    public const int PK_LEN = 1184;
    public const int SK_LEN = 2400;
    public const int CT_LEN = 1088;
    public const int SS_LEN = 32;

    [DllImport(Dll)] static extern int    ppqc_keygen(byte[] pk, byte[] sk);
    [DllImport(Dll)] static extern IntPtr ppqc_encrypt_output_len(IntPtr ptLen);
    [DllImport(Dll)] static extern int    ppqc_encrypt(
        byte[] pk, byte[] pt, IntPtr ptLen, byte[] output, ref IntPtr outLen);
    [DllImport(Dll)] static extern int    ppqc_decrypt(
        byte[] sk, byte[] ct, IntPtr ctLen, byte[] output, ref IntPtr outLen);

    public static (byte[] pk, byte[] sk) GenerateKeyPair()
    {
        var pk = new byte[PK_LEN];
        var sk = new byte[SK_LEN];
        if (ppqc_keygen(pk, sk) ≠ 0)
            throw new Exception("Key generation failed");
        return (pk, sk);
    }

    public static byte[] Encrypt(byte[] pk, string plaintext)
    {
        var pt     = Encoding.UTF8.GetBytes(plaintext);
        var bufLen = ppqc_encrypt_output_len((IntPtr)pt.Length).ToInt64();
        var output = new byte[bufLen];
        var outLen = (IntPtr)bufLen;
        if (ppqc_encrypt(pk, pt, (IntPtr)pt.Length, output, ref outLen) ≠ 0)
            throw new Exception("Encryption failed");
        var result = new byte[outLen.ToInt64()];
        Buffer.BlockCopy(output, 0, result, 0, result.Length);
        return result;
    }

    public static string Decrypt(byte[] sk, byte[] ciphertext)
    {
        var output = new byte[ciphertext.Length + 64];
        var outLen = (IntPtr)output.Length;
        if (ppqc_decrypt(sk, ciphertext, (IntPtr)ciphertext.Length, output, ref
outLen) ≠ 0)
            throw new Exception("Decryption failed — wrong key or tampered data");
        return Encoding.UTF8.GetString(output, 0, outLen.ToInt32());
    }
}

// Usage
class Program
{
```

```
    static void Main()
    {
        var (pk, sk) = PPQC.GenerateKeyPair();
        Console.WriteLine($"PK: {pk.Length} bytes, SK: {sk.Length} bytes");

        byte[] ct  = PPQC.Encrypt(pk, "Hello from C#!");
        string msg = PPQC.Decrypt(sk, ct);
        Console.WriteLine($"Decrypted: {msg}");
    }
}
```

macOS — Swift

Setup

```
MyMacApp/
├── MyMacApp.xcodeproj
├── MyMacApp/
│   ├── p_pqc.h       ← copy here
│   ├── PPQC.swift    ← wrapper file
│   └── MyApp-Bridging-Header.h
└── Libs/
    └── libp_pqc.a    ← copy here
```

Xcode — Configuration

Build Settings → Other Linker Flags:

```
-L$(PROJECT_DIR)/Libs  -lp_pqc
```

Build Settings → Header Search Paths:

```
$(PROJECT_DIR)/MyMacApp
```

Build Settings → Objective-C Bridging Header:

```
MyMacApp/MyApp-Bridging-Header.h
```

Contents of `MyApp-Bridging-Header.h`:

```
#include "p_pqc.h"
```

macOS — Swift code

```
import Foundation

// ── Generate keys ──────────────────────────────────────────────
var pk = Data(count: Int(ppqc_pk_len()))
var sk = Data(count: Int(ppqc_sk_len()))
```

```
let ret = pk.withUnsafeMutableBytes { pkPtr in
    sk.withUnsafeMutableBytes { skPtr in
        ppqc_keygen(
            pkPtr.bindMemory(to: UInt8.self).baseAddress,
            skPtr.bindMemory(to: UInt8.self).baseAddress
        )
    }
}
guard ret == 0 else { fatalError("Key generation failed") }
print("PK: \(pk.count) bytes | SK: \(sk.count) bytes")

// ── Encrypt ──────────────────────────────────────────────────────────
let message  = "Secret message from macOS"
let ptData   = message.data(using: .utf8)!
let capacity = Int(ppqc_encrypt_output_len(ptData.count))
var output   = Data(count: capacity)
var outLen   = capacity

let encRet = pk.withUnsafeBytes { pkPtr in
    ptData.withUnsafeBytes { ptPtr in
        output.withUnsafeMutableBytes { outPtr in
            ppqc_encrypt(
                pkPtr.bindMemory(to: UInt8.self).baseAddress,
                ptPtr.bindMemory(to: UInt8.self).baseAddress,
                ptData.count,
                outPtr.bindMemory(to: UInt8.self).baseAddress,
                &outLen
            )
        }
    }
}
guard encRet == 0 else { fatalError("Encryption failed") }
let blob = output.prefix(outLen)
print("Ciphertext (base64): \(blob.base64EncodedString().prefix(60))...")

// ── Decrypt ──────────────────────────────────────────────────────────
var plain    = Data(count: blob.count + 64)
var plainLen = plain.count

let decRet = sk.withUnsafeBytes { skPtr in
    blob.withUnsafeBytes { ctPtr in
        plain.withUnsafeMutableBytes { outPtr in
            ppqc_decrypt(
                skPtr.bindMemory(to: UInt8.self).baseAddress,
                ctPtr.bindMemory(to: UInt8.self).baseAddress,
                blob.count,
                outPtr.bindMemory(to: UInt8.self).baseAddress,
                &plainLen
            )
        }
    }
}
guard decRet == 0 else { fatalError("Decryption failed") }
print("Decrypted: \(String(data: plain.prefix(plainLen), encoding: .utf8)!)")
```

macOS — Go (CGo)

```
myapp/
├── go.mod
├── main.go
└── ppqc/
    ├── pqc.go          ← CGo wrapper
    ├── p_pqc.h     ← copy here
    └── libp_pqc.a  ← copy here
```

Contents of `ppqc/pqc.go` — the CGo directive portion:

```
/*
#cgo CFLAGS: -I.
#cgo darwin LDFLAGS: ${SRCDIR}/libp_pqc.a -ldl -lpthread \
    -framework Security -framework Foundation
#include "p_pqc.h"
#include <stdlib.h>
*/
import "C"
```

Usage in `main.go`:

```
package main

import (
    "encoding/base64"
    "fmt"
    "log"
    "myapp/ppqc"
)

func main() {
    kp, err := ppqc.GenerateKeyPair()
    if err ≠ nil { log.Fatal(err) }
    fmt.Printf("PK: %d bytes  SK: %d bytes\n",
        len(kp.PublicKey), len(kp.SecretKey))

    ct, err := ppqc.EncryptString(kp.PublicKey, "Hello from Go!")
    if err ≠ nil { log.Fatal(err) }
    fmt.Println("Ciphertext:", ct[:60]+"...")

    msg, err := ppqc.DecryptString(kp.SecretKey, ct)
    if err ≠ nil { log.Fatal(err) }
    fmt.Println("Decrypted:", msg)
    _ = base64.StdEncoding // unused import guard
}
```

```
go build && ./myapp
```

Android — Kotlin (JNI)

Setup

```
app/src/main/
├── java/com/yourapp/
│   └── PPQC.kt       ← wrapper file
└── jniLibs/              ← copy the entire set from the library bundle
    ├── arm64-v8a/
    │   └── libp_pqc.so
    ├── armeabi-v7a/
    │   └── libp_pqc.so
    ├── x86_64/
    │   └── libp_pqc.so
    └── x86/
        └── libp_pqc.so
```

> Android Studio automatically selects the correct ABI when installing on a device — no extra configuration needed.

PPQC.kt — Wrapper

```kotlin
package com.yourapp

import android.util.Base64

object PPQC {

    init { System.loadLibrary("p_pqc") }

    const val PK_LEN = 1184
    const val SK_LEN = 2400
    const val CT_LEN = 1088
    const val SS_LEN = 32

    data class KeyPair(val publicKey: ByteArray, val secretKey: ByteArray) {
        val publicKeyBase64 get() = Base64.encodeToString(publicKey, Base64.NO_WRAP)
        val secretKeyBase64 get() = Base64.encodeToString(secretKey, Base64.NO_WRAP)
    }

    class PQCException(msg: String) : Exception(msg)

    // ── Generate keys ──────────────────────────────────────────
    fun generateKeyPair(): KeyPair {
        val raw = nativeKeygen()
        return KeyPair(
            publicKey = raw.copyOfRange(0, PK_LEN),
            secretKey = raw.copyOfRange(PK_LEN, raw.size)
        )
    }

    // ── Encrypt ────────────────────────────────────────────────
    fun encryptBytes(plaintext: ByteArray, recipientPublicKey: ByteArray): ByteArray {
        require(recipientPublicKey.size == PK_LEN) { "PK phải là $PK_LEN bytes" }
        return nativeEncryptBytes(recipientPublicKey, plaintext) ?: throw
```

```
PQCException("Encryption failed")
    }

    fun encrypt(plaintext: String, recipientPublicKey: ByteArray): String =
        Base64.encodeToString(encryptBytes(plaintext.toByteArray(Charsets.UTF_8),
recipientPublicKey), Base64.NO_WRAP)

    // ── Decrypt ──────────────────────────────────────────────────────────
    fun decryptBytes(blob: ByteArray, secretKey: ByteArray): ByteArray {
        require(secretKey.size == SK_LEN) { "SK phải là $SK_LEN bytes" }
        return nativeDecryptBytes(secretKey, blob) ?: throw PQCException("Decryption
failed")
    }

    fun decrypt(ciphertextBase64: String, secretKey: ByteArray): String =
        String(decryptBytes(Base64.decode(ciphertextBase64, Base64.NO_WRAP),
secretKey), Charsets.UTF_8)

    // ── KEM ──────────────────────────────────────────────────────────────
    fun kemEncapsulate(publicKey: ByteArray): Pair<ByteArray, ByteArray> {
        val raw = nativeKemEncapsulate(publicKey)
        return raw.copyOfRange(0, CT_LEN) to raw.copyOfRange(CT_LEN, raw.size)
    }

    fun kemDecapsulate(secretKey: ByteArray, ct: ByteArray): ByteArray =
        nativeKemDecapsulate(secretKey, ct)

    // ── Restore keys from base64 ─────────────────────────────────────────
    fun keyPairFromBase64(pkB64: String, skB64: String) = KeyPair(
        publicKey = Base64.decode(pkB64, Base64.NO_WRAP),
        secretKey = Base64.decode(skB64, Base64.NO_WRAP)
    )

    private external fun nativeKeygen(): ByteArray
    private external fun nativeEncryptBytes(pk: ByteArray, plaintext: ByteArray):
ByteArray?
    private external fun nativeDecryptBytes(sk: ByteArray, blob: ByteArray):
ByteArray?
    private external fun nativeKemEncapsulate(pk: ByteArray): ByteArray
    private external fun nativeKemDecapsulate(sk: ByteArray, ct: ByteArray): ByteArray
}
```

Usage in an Activity / ViewModel

```
import com.yourapp.PPQC

// Generate keys (Bob — the recipient)
val kp = PPQC.generateKeyPair()

// Share the publicKey to the server for Alice
uploadToServer(kp.publicKeyBase64)

// Alice encrypts and sends to Bob
val bobPK   = Base64.decode(getPKFromServer(), Base64.NO_WRAP)
val cipher  = PPQC.encrypt("Secret content", bobPK)
sendToServer(cipher)
```

```
// Bob decrypts
val message = PPQC.decrypt(receivedCipher, kp.secretKey)
println("Bob received: $message")
```

Storing keys securely — Android Keystore

```
import android.security.keystore.KeyGenParameterSpec
import android.security.keystore.KeyProperties
import javax.crypto.Cipher
import javax.crypto.KeyGenerator
import javax.crypto.spec.GCMParameterSpec

// Wrap the secret key with AES-GCM inside the Android Keystore
fun saveSecretKey(context: Context, secretKey: ByteArray) {
    val kg = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
"AndroidKeyStore")
    kg.init(KeyGenParameterSpec.Builder("p_pqc_key",
        KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
        .build())
    val wrapKey = kg.generateKey()

    val cipher = Cipher.getInstance("AES/GCM/NoPadding")
    cipher.init(Cipher.ENCRYPT_MODE, wrapKey)
    val encrypted = cipher.doFinal(secretKey)

    context.getSharedPreferences("pqc", 0).edit()
        .putString("sk", Base64.encodeToString(encrypted, Base64.NO_WRAP))
        .putString("iv", Base64.encodeToString(cipher.iv,  Base64.NO_WRAP))
        .apply()
}
```

---

iOS — Swift Native

Setup

```
MyiOSApp.xcodeproj
├── MyiOSApp/
│   ├── MyApp-Bridging-Header.h
│   ├── PPQC.swift       ← wrapper file
│   └── ...
└── Frameworks/
    └── PPQC.xcframework ← drag in here
```

Xcode — Configuration

1. Drag `PPQC.xcframework` into the **Frameworks** folder in the Xcode navigator
2. **Target → General → Frameworks, Libraries → Add → PPQC.xcframework**
3. Choose **"Do Not Embed"** (static library)
4. **Build Settings → Objective-C Bridging Header:** `MyiOSApp/MyApp-Bridging-Header.h`

Contents of `MyApp-Bridging-Header.h`:

```
#include <p_pqc.h>
```

iOS — Swift wrapper

```
import Foundation

public enum PPQC {

    public static let pkLen = Int(ppqc_pk_len())
    public static let skLen = Int(ppqc_sk_len())

    // ── Generate keys ──────────────────────────────────────────────
    public static func generateKeyPair() throws → (pk: Data, sk: Data) {
        var pk = Data(count: pkLen)
        var sk = Data(count: skLen)
        let r = pk.withUnsafeMutableBytes { p in
            sk.withUnsafeMutableBytes { s in
                ppqc_keygen(p.baseAddress?.assumingMemoryBound(to: UInt8.self),
                            s.baseAddress?.assumingMemoryBound(to: UInt8.self))
            }
        }
        guard r == 0 else { throw PpqcError.keygenFailed }
        return (pk, sk)
    }

    // ── Encrypt → base64 ──────────────────────────────────────────
    public static func encrypt(_ text: String, publicKey pk: Data) throws → String {
        guard let pt = text.data(using: .utf8) else {
            throw PpqcError.encryptFailed
        }
        let cap   = Int(ppqc_encrypt_output_len(pt.count))
        var out   = Data(count: cap)
        var outLen = cap
        let r = pk.withUnsafeBytes { pkPtr in
            pt.withUnsafeBytes { ptPtr in
                out.withUnsafeMutableBytes { outPtr in
                    ppqc_encrypt(
                        pkPtr.baseAddress?.assumingMemoryBound(to: UInt8.self),
                        ptPtr.baseAddress?.assumingMemoryBound(to: UInt8.self),
                        pt.count,
                        outPtr.baseAddress?.assumingMemoryBound(to: UInt8.self),
                        &outLen)
                }
            }
        }
        guard r == 0 else { throw PpqcError.encryptFailed }
        return out.prefix(outLen).base64EncodedString()
    }

    // ── Decrypt ────────────────────────────────────────────────────
    public static func decrypt(_ b64: String, secretKey sk: Data) throws → String {
        guard let blob = Data(base64Encoded: b64) else {
            throw PpqcError.invalidInput
        }
```

```
        var out    = Data(count: blob.count + 64)
        var outLen = out.count
        let r = sk.withUnsafeBytes { skPtr in
            blob.withUnsafeBytes { ctPtr in
                out.withUnsafeMutableBytes { outPtr in
                    ppqc_decrypt(
                        skPtr.baseAddress?.assumingMemoryBound(to: UInt8.self),
                        ctPtr.baseAddress?.assumingMemoryBound(to: UInt8.self),
                        blob.count,
                        outPtr.baseAddress?.assumingMemoryBound(to: UInt8.self),
                        &outLen)
                }
            }
        }
        guard r == 0 else { throw PpqcError.decryptFailed }
        return String(data: out.prefix(outLen), encoding: .utf8) ?? ""
    }

    public enum PpqcError: Error {
        case keygenFailed, encryptFailed, decryptFailed, invalidInput
    }
}
```

Usage in a ViewController / SwiftUI

```
// Generate keys
let (pk, sk) = try PPQC.generateKeyPair()
print("PK: \(pk.count)B  SK: \(sk.count)B")

// Encrypt
let cipher = try PPQC.encrypt("Hello iOS!", publicKey: pk)

// Decrypt
let plain  = try PPQC.decrypt(cipher, secretKey: sk)
print(plain)  // → Hello iOS!
```

Storing keys — iOS Keychain

```
import Security

func saveToKeychain(_ data: Data, key: String) {
    let q: [CFString: Any] = [
        kSecClass:       kSecClassGenericPassword,
        kSecAttrAccount: key,
        kSecValueData:   data,
        kSecAttrAccessible: kSecAttrAccessibleWhenUnlockedThisDeviceOnly
    ]
    SecItemDelete(q as CFDictionary)
    SecItemAdd(q as CFDictionary, nil)
}

func loadFromKeychain(key: String) -> Data? {
    let q: [CFString: Any] = [
        kSecClass:       kSecClassGenericPassword,
        kSecAttrAccount: key,
        kSecReturnData:  true,
```

```
        kSecMatchLimit:  kSecMatchLimitOne
    ]
    var result: AnyObject?
    SecItemCopyMatching(q as CFDictionary, &result)
    return result as? Data
}

// Save
saveToKeychain(sk, key: "ppqc.pqc.secretkey")

// Load
let secretKey = loadFromKeychain(key: "ppqc.pqc.secretkey")!
```

Flutter — dart:ffi (iOS + Android)

Flutter uses `dart:ffi` to call the C API directly — not through JNI.

Setup

iOS: Copy `PPQC.xcframework` into `ios/Frameworks/`, open `ios/Runner.xcworkspace`:
- Drag the framework into the **Frameworks group**
- **Build Phases → Link Binary → Add → PPQC.xcframework**
- Embed: **Do Not Embed**

Android: Copy the entire `jniLibs/` into `android/app/src/main/jniLibs/`

pubspec.yaml:

```
dependencies:
  ffi: ^2.1.0
```

```
flutter pub get
```

p_pqc.dart — dart:ffi wrapper

```
import 'dart:ffi';
import 'dart:io';
import 'dart:convert';
import 'dart:typed_data';

// ── Load the library per platform ──────────────────────────────
final _lib = () {
  if (Platform.isAndroid) return DynamicLibrary.open('libp_pqc.so');
  if (Platform.isIOS)     return DynamicLibrary.process();
  if (Platform.isMacOS)   return DynamicLibrary.open('libp_pqc.dylib');
  if (Platform.isWindows) return DynamicLibrary.open('p_pqc.dll');
  throw UnsupportedError(Platform.operatingSystem);
}();

// ── FFI bindings ───────────────────────────────────────────────
final _keygen  = _lib.lookupFunction<
    Int32 Function(Pointer<Uint8>, Pointer<Uint8>),
```

```
    int   Function(Pointer<Uint8>, Pointer<Uint8>)>('ppqc_keygen');

final _encLen  = _lib.lookupFunction<
    Size Function(Size), int Function(int)>('ppqc_encrypt_output_len');

final _encrypt = _lib.lookupFunction<
    Int32 Function(Pointer<Uint8>, Pointer<Uint8>, Size, Pointer<Uint8>,
Pointer<Size>),
    int   Function(Pointer<Uint8>, Pointer<Uint8>, int,  Pointer<Uint8>,
Pointer<Size>)
>('ppqc_encrypt');

final _decrypt = _lib.lookupFunction<
    Int32 Function(Pointer<Uint8>, Pointer<Uint8>, Size, Pointer<Uint8>,
Pointer<Size>),
    int   Function(Pointer<Uint8>, Pointer<Uint8>, int,  Pointer<Uint8>,
Pointer<Size>)
>('ppqc_decrypt');

// ── Constants ──────────────────────────────────────────────────────────
const pkLen = 1184;
const skLen = 2400;

// ── PPQC API ───────────────────────────────────────────────────────────
class PPQC {

  /// Generate a Kyber-768 key pair
  static ({Uint8List pk, Uint8List sk}) generateKeyPair() {
    final pkPtr = calloc<Uint8>(pkLen);
    final skPtr = calloc<Uint8>(skLen);
    try {
      if (_keygen(pkPtr, skPtr) != 0) throw Exception('Keygen failed');
      return (
        pk: Uint8List.fromList(pkPtr.asTypedList(pkLen)),
        sk: Uint8List.fromList(skPtr.asTypedList(skLen)),
      );
    } finally {
      calloc.free(pkPtr); calloc.free(skPtr);
    }
  }

  /// Encrypt — returns a base64 blob
  static String encrypt(String plaintext, Uint8List publicKey) {
    final pt    = utf8.encode(plaintext);
    final cap   = _encLen(pt.length);
    final pkPtr = calloc<Uint8>(pkLen);
    final ptPtr = calloc<Uint8>(pt.length);
    final outPtr= calloc<Uint8>(cap);
    final lenPtr= calloc<Size>();
    try {
      pkPtr.asTypedList(pkLen).setAll(0, publicKey);
      ptPtr.asTypedList(pt.length).setAll(0, pt);
      lenPtr.value = cap;
      if (_encrypt(pkPtr, ptPtr, pt.length, outPtr, lenPtr) != 0)
        throw Exception('Encryption failed');
      return base64.encode(outPtr.asTypedList(lenPtr.value));
    } finally {
      calloc.free(pkPtr); calloc.free(ptPtr);
      calloc.free(outPtr); calloc.free(lenPtr);
```

```
    }
  }

  /// Decrypt — takes a base64 blob, returns a string
  static String decrypt(String ciphertextB64, Uint8List secretKey) {
    final ct    = base64.decode(ciphertextB64);
    final skPtr = calloc<Uint8>(skLen);
    final ctPtr = calloc<Uint8>(ct.length);
    final outPtr= calloc<Uint8>(ct.length + 64);
    final lenPtr= calloc<Size>();
    try {
      skPtr.asTypedList(skLen).setAll(0, secretKey);
      ctPtr.asTypedList(ct.length).setAll(0, ct);
      lenPtr.value = ct.length + 64;
      if (_decrypt(skPtr, ctPtr, ct.length, outPtr, lenPtr) != 0)
        throw Exception('Decryption failed — wrong key or modified data');
      return utf8.decode(outPtr.asTypedList(lenPtr.value));
    } finally {
      calloc.free(skPtr); calloc.free(ctPtr);
      calloc.free(outPtr); calloc.free(lenPtr);
    }
  }
}
```

Usage in a Widget

```
final kp    = PPQC.generateKeyPair();
final ct    = PPQC.encrypt('Hello Flutter!', kp.pk);
final plain = PPQC.decrypt(ct, kp.sk);
print(plain);  // → Hello Flutter!
```

React Native — JSI / NativeModule

For React Native, create a Native Module that calls the C API:

Android — `PpqcModule.kt`:

```
class PpqcModule(reactContext: ReactApplicationContext)
    : ReactContextBaseJavaModule(reactContext) {

    override fun getName() = "PPQC"

    @ReactMethod
    fun generateKeyPair(promise: Promise) {
        val kp = PPQC.generateKeyPair()
        promise.resolve(Arguments.makeNativeMap(mapOf(
            "pk" to kp.publicKeyBase64,
            "sk" to kp.secretKeyBase64
        )))
    }

    @ReactMethod
    fun encrypt(plaintext: String, pkB64: String, promise: Promise) {
        val pk = Base64.decode(pkB64, Base64.NO_WRAP)
        promise.resolve(PPQC.encrypt(plaintext, pk))
    }
```

```
    @ReactMethod
    fun decrypt(ciphertext: String, skB64: String, promise: Promise) {
        val sk = Base64.decode(skB64, Base64.NO_WRAP)
        promise.resolve(PPQC.decrypt(ciphertext, sk))
    }
}
```

JavaScript / TypeScript:

```
import { NativeModules } from 'react-native';
const { PPQC } = NativeModules;

const { pk, sk } = await PPQC.generateKeyPair();
const cipher      = await PPQC.encrypt('Hello PQC!', pk);
const plain       = await PPQC.decrypt(cipher, sk);
console.log(plain);
```

WASM — Browser / JavaScript / TypeScript

p-pqc runs directly in the browser via WebAssembly — no backend required, nothing extra to install.

Building the WASM library

```
# Install wasm-pack (once)
brew install wasm-pack        # macOS
# or: cargo install wasm-pack

# Build → dist/web/
./build.sh wasm
```

Output in `dist/web/` :

```
dist/web/
├── p_pqc.js          ← ES module loader + glue code
├── p_pqc_bg.wasm     ← binary (~80 KB)
├── p_pqc.d.ts        ← TypeScript type definitions
├── p_pqc_bg.wasm.d.ts
└── example.html          ← a working demo
```

> **Important:** You must serve it over HTTP (do not open file:// directly). The reason: browsers block `fetch()` over the `file://` protocol when loading WASM.

```
# Run a quick demo
cd dist/web && python3 -m http.server 8080
# Open: http://localhost:8080/example.html
```

WASM API — Overview

```
import init, {
  keygen,          // generate a Kyber-768 key pair
  encrypt,         // encrypt with a public key → wire-format blob
  decrypt,         // decrypt with a secret key
  kem_encapsulate, // KEM: derive a shared secret from a public key
  kem_decapsulate, // KEM: recover the shared secret
  pk_len,          // 1184
  sk_len,          // 2400
  ct_len,          // 1088
  ss_len,          // 32
} from "./p_pqc.js";

await init(); // REQUIRED — call once before using any function
```

Sizes and data types

Function / Property	Return type	Size
`keygen().publicKey`	`Uint8Array`	1184 bytes
`keygen().secretKey`	`Uint8Array`	2400 bytes
`encrypt(pk, pt)`	`Uint8Array`	pt.length + 1121 bytes
`decrypt(sk, blob)`	`Uint8Array`	original plaintext
`kem_encapsulate(pk).ciphertext`	`Uint8Array`	1088 bytes
`kem_encapsulate(pk).sharedSecret`	`Uint8Array`	32 bytes
`kem_decapsulate(sk, ct)`	`Uint8Array`	32 bytes

Example — Plain HTML (ES Module)

```
<!DOCTYPE html>
<html>
<head><meta charset="utf-8"><title>p-pqc demo</title></head>
<body>
<script type="module">
import init, { keygen, encrypt, decrypt } from "./p_pqc.js";

await init();  // load WASM

const enc = new TextEncoder();
const dec = new TextDecoder();

// 1. Generate a key pair
const kp = keygen();
console.log("PK:", kp.publicKey.length, "bytes");
console.log("SK:", kp.secretKey.length, "bytes");

// 2. Encrypt
const plaintext = enc.encode("Hello from p-pqc WASM!");
```

```
const blob = encrypt(kp.publicKey, plaintext);
console.log("Ciphertext:", blob.length, "bytes");

// 3. Decrypt
const recovered = decrypt(kp.secretKey, blob);
console.log("Decrypted:", dec.decode(recovered));
</script>
</body>
</html>
```

Example — React / Vite / Webpack

Install the package:

```
npm install ./dist/web   # or copy dist/web into src/lib/p-pqc/
```

React component:

```
import { useEffect, useState } from "react";

// Lazy-load the WASM module
let _ppqc: typeof import("p-pqc") | null = null;
async function loadPpqc() {
  if (_ppqc) return _ppqc;
  const mod = await import("p-pqc");
  await mod.default();   // init()
  _ppqc = mod;
  return mod;
}

export function usePQC() {
  const [ready, setReady] = useState(false);

  useEffect(() => {
    loadPpqc().then(() => setReady(true));
  }, []);

  const generateKeyPair = async () => {
    const mod = await loadPpqc();
    const kp  = mod.keygen();
    return {
      publicKey:  kp.publicKey,   // Uint8Array 1184 bytes
      secretKey:  kp.secretKey,   // Uint8Array 2400 bytes
    };
  };

  const encryptMsg = async (message: string, publicKey: Uint8Array) => {
    const mod = await loadPpqc();
    const pt  = new TextEncoder().encode(message);
    return mod.encrypt(publicKey, pt);  // Uint8Array blob
  };

  const decryptMsg = async (blob: Uint8Array, secretKey: Uint8Array) => {
    const mod      = await loadPpqc();
    const recovered = mod.decrypt(secretKey, blob);
    return new TextDecoder().decode(recovered);
  };
```

```
  return { ready, generateKeyPair, encryptMsg, decryptMsg };
}
```

Usage in a component:

```
function EncryptDemo() {
  const { ready, generateKeyPair, encryptMsg, decryptMsg } = usePQC();

  const handleDemo = async () => {
    const kp      = await generateKeyPair();
    const blob    = await encryptMsg("Secret message", kp.publicKey);
    const message = await decryptMsg(blob, kp.secretKey);
    console.log("Roundtrip OK:", message);
  };

  return <button disabled={!ready} onClick={handleDemo}>Try PQC</button>;
}
```

Example — Raw KEM (manage the symmetric key yourself)

```
import init, { keygen, kem_encapsulate, kem_decapsulate } from "./p_pqc.js";

await init();

// Alice generates keys, shares her publicKey to the server
const alice = keygen();

// Bob takes Alice's publicKey, derives a shared secret
const enc = kem_encapsulate(alice.publicKey);
//   enc.ciphertext    → 1088 bytes (sent to Alice)
//   enc.sharedSecret  → 32 bytes   (Bob keeps it secret — use as an AES/ChaCha key)

// Alice recovers the shared secret
const ss = kem_decapsulate(alice.secretKey, enc.ciphertext);
//   ss → 32 bytes, identical to enc.sharedSecret

const match = enc.sharedSecret.every((b, i) => b === ss[i]);
console.log("Shared secret matches:", match);  // → true

// Both sides now have the same 32-byte secret to use with AES-GCM or ChaCha20
```

Vite — optimized configuration

vite.config.ts:

```
import { defineConfig } from "vite";

export default defineConfig({
  optimizeDeps: {
    exclude: ["p-pqc"],  // do not transpile the WASM module
  },
  server: {
    headers: {
      // SharedArrayBuffer requires COOP + COEP (for Atomics if you use a Worker)
```

```
      "Cross-Origin-Opener-Policy":   "same-origin",
      "Cross-Origin-Embedder-Policy": "require-corp",
    },
  },
});
```

Next.js — Dynamic Import

```
// lib/ppqc.ts — runs client-side only
export async function getPpqc() {
  if (typeof window === "undefined") {
    throw new Error("p-pqc only runs on the client side");
  }
  const mod = await import("p-pqc");
  await mod.default();  // init WASM
  return mod;
}

// In a component
const { keygen, encrypt, decrypt } = await getPpqc();
```

Node.js

```
import { readFile } from "fs/promises";
import { fileURLToPath } from "url";
import { dirname, join } from "path";

// Node 18+ supports WASM natively
const __dir   = dirname(fileURLToPath(import.meta.url));
const wasmBuf = await readFile(join(__dir, "p_pqc_bg.wasm"));

import initSync, { keygen, encrypt, decrypt } from "./p_pqc.js";
await initSync({ module: new WebAssembly.Module(wasmBuf) });

const kp      = keygen();
const pt      = Buffer.from("Hello from Node.js");
const blob    = encrypt(kp.publicKey, pt);
const plain   = decrypt(kp.secretKey, blob);
console.log(Buffer.from(plain).toString("utf8"));
```

Storing keys — Browser

```
// Store in IndexedDB (do not use localStorage — not secure enough)
async function saveKeyPair(pk, sk) {
  const db = await openDB("ppqc-keys", 1, {
    upgrade(db) { db.createObjectStore("keys"); }
  });
  // pk can be stored freely; sk should be additionally encrypted with the Web Crypto
API
  await db.put("keys", pk, "publicKey");

  // Encrypt sk with Web Crypto before storing
  const wrappingKey = await crypto.subtle.generateKey(
    { name: "AES-GCM", length: 256 }, false, ["wrapKey", "unwrapKey"]
```

```
  );
  // ... (see the Web Crypto API docs to wrap/unwrap the key)
  await db.put("keys", sk, "secretKey_encrypted");
}
```

WASM performance (measured on Chrome, M3 MacBook)

Operation	Time	Notes
Load WASM (`init()`)	~5 ms	Called once only
`keygen()`	~1.2 ms	Generate a Kyber-768 keypair
`encrypt()` (1KB)	~1.5 ms	KEM + AES-256-GCM
`decrypt()` (1KB)	~1.0 ms	Decap + AEAD verify
`kem_encapsulate()`	~1.0 ms	KEM only
`kem_decapsulate()`	~0.8 ms	KEM decap only

> WASM is ~2–3× slower than native but still fast enough for every real-world application.

Wire Format — Cross-platform compatibility

Ciphertext produced by **any platform** can be decrypted by any other platform.

```
Offset    Size         Content
──────    ──────────   ──────────────────────────────────────────
0         1 byte       Version: 0x06 (0x05, 0x04 legacy — decrypt only)
1-4       4 bytes      KEM ciphertext length (little-endian) = 1088
5-1092    1088 bytes   ML-KEM-768 KEM ciphertext
1093-1104 12 bytes     Random 96-bit nonce (HMAC_DRBG)
1105-N    N bytes      Ciphertext (encrypted data)
N+1-N+16  16 bytes     Authentication tag (GCM; header là AAD)
```

Fixed overhead: 1121 bytes for every message.

Plaintext size	Blob size	Base64
10 bytes	1131 bytes	1508 characters
100 bytes	1221 bytes	1628 characters
1000 bytes	2121 bytes	2828 characters
10000 bytes	11121 bytes	14828 characters

Key Sizes and Performance

Parameter	Value	Notes
Public key	1184 bytes	Share freely
Secret key	2400 bytes	Must be stored securely
KEM ciphertext	1088 bytes	Inside the wire format
Shared secret	32 bytes	Use as an AES/ChaCha key
Keygen	~0.5 ms	ARM64
Encrypt	~0.8 ms	ARM64, 1KB payload
Decrypt	~0.6 ms	ARM64, 1KB payload

Security — Storing Keys Correctly

Public key — can be stored freely:

```
SharedPreferences / UserDefaults / Database / Server
```

Secret key — must use secure storage:

Platform	Required storage location
iOS	Keychain — `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`
Android	Android Keystore — wrapped with AES-GCM
Flutter	flutter_secure_storage package
Windows	Windows DPAPI or Windows Credential Manager
macOS	Keychain Services
Go / Server	Vault, AWS KMS, Azure Key Vault, or an HSM

Absolutely never:

```
✗  Store the secret key in SharedPreferences / UserDefaults as plain text
✗  Log the secret key to the console
✗  Send the secret key over the network
✗  Hard-code the secret key in source code
```

Security Hardening

The **2026-10-15** update fixes two findings of the technical review (no API change, no blob format change):

- **No division on secret data (KyberSlash class):** `qmod` (`a % Q`), ciphertext compression (`(x·2^d + Q/2) / Q`) and message decoding are now hand-written multiply-and-shift (Barrett; constants 80635 / 1290167 as in the pq-crystals fix), bit-identical to the old formulas (exhaustive tests). Previously only native release builds were rewritten by LLVM; the **WASM** build still contained `i32.div_s` / `i32.rem_s`, which a browser may JIT into variable-time hardware division, and debug builds contained `sdiv`. `tests/ct/no-div.sh` disassembles the WASM release build and the host debug build and fails if any `p_pqc::*` function contains a division (sole exception `p_pqc::keccak`, which only divides public lengths/indices).
- **Fork-safe DRBG:** on unix each thread's HMAC_DRBG records the PID it was seeded in; when the PID changes (we are in a child process) it reseeds from fresh OS entropy before generating, so a child no longer repeats the parent's encapsulation `m` or GCM nonces. The `drbg_reseeds_after_fork` test really forks and compares parent/child bytes.

The following hardening fixes were applied to the library core (2026-06-21) to remove side-channel and denial-of-service risks. They strengthen the implementation without changing any cryptographic algorithm or output format.

- **Constant-time FO transform:** the Fujisaki-Okamoto re-encryption comparison in decapsulation now uses `subtle::ConstantTimeEq` with a branchless bitwise select instead of a data-dependent `if`, eliminating a timing oracle that could break IND-CCA2 security.
- **DoS hardening:** `pqc_decrypt` now validates the embedded KEM ciphertext length (`kem_len == CT_BYTES`) before decapsulation, rejecting malformed blobs that previously caused a panic.
- **Memory zeroization:** `KeyPair` and `Encapsulation` now derive `Zeroize` / `zeroize(drop)`, and all intermediate secret values (keygen seeds d/z, KEM message m, hash buffer g, decrypted m_prime, intermediate secret key) are zeroized after use.
- **JNI boundary validation:** the Android JNI bridge now validates public-key / secret-key / ciphertext lengths before calling into the Kyber core (a short Java `ByteArray` previously caused a panic that aborts the whole app on Rust 1.81+), and replaced `.unwrap()` calls with graceful null returns to avoid aborts on NUL-byte plaintext or JVM OOM.
- **Note:** the verified-correct crypto core (Keccak/SHA-3, SHAKE, NTT, ChaCha20-Poly1305 AEAD) was unchanged.

> The public C ABI and API signatures did **not** change, so no integration code changes are required.

Contact

Email: letanphuoc@gmail.com

Library: `p-pqc`
Header: `p_pqc.h`
Full examples: `examples/swift/` · `examples/kotlin/` · `examples/flutter/`