Post quantum cryptography in .NET

2023/02/21 · 1206 words · 6 minutes to read

security quantum computing c-sharp net 7

I have written extensively about quantum computing on this blog before. Quantum computing has the potential to break many of the cryptographic systems that we use today. Shor's algorithm, for example, can efficiently factor large numbers, which would make widely-used asymmetric cryptography schemes such as RSA and elliptic curves insecure.

In this post, we'll explore how to use post-quantum cryptography from a C# program, using CRYSTALS-Kyber and CRYSTALS-Dilithium as examples.

Background

To address the quantum computing threats, in 2016, the National Institute of Standards and Technology (NIST) launched a post-quantum cryptography competition to develop standardized cryptographic schemes that are resistant to quantum attacks.

In the summer of 2022, NIST announced the first four winners of the competition, which include CRYSTALS-Kyber for asymmetric encryption, CRYSTALS-Dilithium and FALCON for signatures, and SPHINCS+ for stateless hash-based signatures.

Kyber is a cryptographic scheme that provides key sharing/encapsulation functionality, whose security rely on computational difficulty related to solving problems related to the mathematical abstract structures of lattices. For public key cryptography, lattices rely on finding a short vector in a high dimensional lattice. Dilithium is a cryptographic scheme that provides signature verification functionality and, like Kyber, it uses lattices to generate keys and signatures.

The reference implementations for CRYSTALS algorithms are done in C, but there exist versions in many other languages too, such as Go (maintained by Cloudflare) or Rust.

In November 2022, the popular BouncyCastle cryptography library, available for Java and .NET, published a big update to their .NET variant. It encompassed major version change to 2.0.0, adoption of semantic versioning, creation of a new official Nuget package, moving to .NET 6.0 (while still cross compiling to .NET Framework and .NET Standard 2.0) and - most interestingly to us, providing implementations of NIST Post-Quantum Cryptography Standardization algorithms (not only the four winners but of several others as well). This gives us Kyber and Dilithium implementations for the .NET ecosystem.

Getting started

To get going we need to create a new .NET console application and add a reference to `BouncyCastle.Cryptography`.

```
mkdir dotnet-kyber-dilithium-demo
cd dotnet-kyber-dilithium-demo
dotnet new console
dotnet package add BouncyCastle.Cryptography
```

This should set us up with the latest (current) version of the package, 2.1.1, and an empty .NET 7 console app.

We will then define a simple helper method to print out the contents of the byte array as base 64 encoded string. This will allow us to easily output the different artifacts created at various steps in our code, which in turn helps us to trace how the protocol plays out. For long byte arrays there is no need to print them in their entirety, so we shall truncate them to 25 characters at the start and end.

```
static string PrettyPrint(byte[] bytes) {
    var base64 = Convert.ToBase64String(bytes);
    if (base64.Length > 50)
        return $"{base64[..25]}...{base64[^25..]}";

    return base64;
}
```

Dilithium

Let's start our exploration with Dilithium - so signature verification. First, we will create the message we would like to sign and later verify:

```
var data = Hex.Encode(Encoding.ASCII.GetBytes("Hello, Dilithium!")
Console.WriteLine($"Message: {PrettyPrint(data)}");
```

Next, we need to initialize `DilithiumKeyGenerationParameters` with the desired algorithm strength and a random number generator. For demo purposes, the `SecureRandom` built-in BouncyCastle will suffice, but ideally this would be tied to hardware generated numbers. We will use `Dilithium3` here which offers security level comparable to 128-bit AES against both classical and quantum threats.

```
var random = new SecureRandom();
var keyGenParameters = new DilithiumKeyGenerationParameters(randor
```

Once the parameters are available, a key pair can be generated using `DilithiumKeyPairGenerator`:

```
var dilithiumKeyPairGenerator = new DilithiumKeyPairGenerator();
dilithiumKeyPairGenerator.Init(keyGenParameters);
var keyPair = dilithiumKeyPairGenerator.GenerateKeyPair();
```

At this stage, we can view the public and private keys. The private will be used for signing and the public for verifying the signature.

```
// get and view the keys
var publicKey = (DilithiumPublicKeyParameters)keyPair.Public;
var privateKey = (DilithiumPrivateKeyParameters)keyPair.Private;
var pubEncoded = publicKey.GetEncoded();
var privateEncoded = privateKey.GetEncoded();
Console.WriteLine($"Public key: {PrettyPrint(pubEncoded)}");
Console.WriteLine($"Private key: {PrettyPrint(privateEncoded)}");
```

Next, we can sign our message. In a good old-fashioned custom of security, we shall refer to the signer as “Alice”. Alice can use the type `DilithiumSigner` and her private key to create the signature - when the type gets initialized with the key, the first boolean parameter is used to specify whether we shall use the instance of the `DilithiumSigner` for signing or not.

The signature is then printed to the console.

```
// sign
var alice = new DilithiumSigner();
alice.Init(true, privateKey);
var signature = alice.GenerateSignature(data);
Console.WriteLine($"Signature: {PrettyPrint(signature)}");
```

Finally, the counterpart of Alice, Bob, can verify the signature. For that, he initializes his own `DilithiumSigner`, this time passing `false` as the first parameter, and using the public key.

The method `VerifySignature` on `DilithiumSigner` returns a boolean indicating if the signature was successfully validated. Had anyone tampered with the contents, this verification would fail.

```
// verify signature
var bob = new DilithiumSigner();
bob.Init(false, publicKey);
var verified = bob.VerifySignature(data, signature);
Console.WriteLine($"Successfully verified? {verified}");
Console.WriteLine("");
```

If we now run this program, the output should resemble this:

```
Message: NDg2NTZjNmM2ZjJjMjA0NDY5NmM2OTc0Njg2OTc1NmQyMQ==
Public key: BN+aZrE0o6NatVFlykF9qKMGq...xmOoAUFtJ9RXfPfYcG2Ui2vM=
Private key: BN+aZrE0o6NatVFlykF9qKMGq...HO+lj9KInVKeYgK3OkZJPCA==
Signature: Qcvm23Qibndu0qTqRL7SjRhiy...AAAAAAAAAAAAAAAFCQ4SGCA=
Successfully verified? True
```

KYBER

Using very similar steps (kudos to Bouncy Castle API designers!), we can demonstrate the usage of CRYSTALS-Kyber for key encapsulation. This way, Alice and Bob can securely exchange a secret which they can then use for standard symmetric encryption.

We start by initalizing a random number generator and `KyberKeyGenerationParameters` . The recommended default for Kyber is to use the 768-bit variant, though it also comes with 512 and 1024-bit variants. The 768-bit variant achieves 128-bit security against all known classical and quantum attacks.

```
var random = new SecureRandom();
var keyGenParameters = new KyberKeyGenerationParameters(random, Ky
```

Next a key pair is generated:

```
var kyberKeyPairGenerator = new KyberKeyPairGenerator();
kyberKeyPairGenerator.Init(keyGenParameters);

// generate key pair for Alice
var aliceKeyPair = kyberKeyPairGenerator.GenerateKeyPair();
```

The Kyber keys - both public and private can be viewed at this point. Alice keeps here private key secret, while the public one can be announced:

```
// get and view the keys
var alicePublic = (KyberPublicKeyParameters)aliceKeyPair.Public;
var alicePrivate = (KyberPrivateKeyParameters)aliceKeyPair.Private
var pubEncoded = alicePublic.GetEncoded();
var privateEncoded = alicePrivate.GetEncoded();
Console.WriteLine($"Alice's Public key: {PrettyPrint(pubEncoded)}"
Console.WriteLine($"Alice's Private key: {PrettyPrint(privateEnco
```

Using Alice’s public key, Bob can use `KyberKemGenerator` (again, initalized with a random number generator) to generate and encapsulate a shared secret. That secret can be extracted and printed for our convenience:

```
// Bob encapsulates a new shared secret using Alice's public key
var bobKyberKemGenerator = new KyberKemGenerator(random);
var encapsulatedSecret = bobKyberKemGenerator.GenerateEncapsulate
var bobSecret = encapsulatedSecret.GetSecret();
Console.WriteLine($"Bob's Secret: {PrettyPrint(bobSecret)}");
```

The ciphertext can then be passed over to Alice:

```
// cipher text produced by Bob and sent to Alice
var cipherText = encapsulatedSecret.GetEncapsulation();
Console.WriteLine($"Cipher text: {PrettyPrint(cipherText)}");
```

Alice can decapsulate the shared secret using her own instance of `KyberKemGenerator` and her matching private key:

```
// Alice decapsulates a new shared secret using Alice's private ke
var aliceKemExtractor = new KyberKemExtractor(alicePrivate);
var aliceSecret = aliceKemExtractor.ExtractSecret(cipherText);
Console.WriteLine($"Alice's Secret: {PrettyPrint(aliceSecret)}");
```

At this stage Alice and Bob should have a shared secret in their possession and can use that for communication secured with symmetric cryptography:

```
// Compare secrets
var equal = bobSecret.SequenceEqual(aliceSecret);
Console.WriteLine($"Secrets equal? {equal}");
Console.WriteLine("");
```

Upon execution, we should see the following:

```
Alice's Public key: TxIujEemfoJdyEdzJwLFoCLLE...vk+MD7tzGmnUChzSpu
Alice's Private key: qZU4LXsi/BVQOdAqlQcq4nN7d...LWvS0x65u/jJ+slDz
Bob's Secret: hqKSqCzy+4FRfM57qgOdxRnoISNF7dCRfa9Z82Tyuzo=
Cipher text: dlzusNveTI8Cp1xa87vCE20aD...ceq57qIM3259oNRUrKpuhjk4=
Alice's Secret: hqKSqCzy+4FRfM57qgOdxRnoISNF7dCRfa9Z82Tyuzo=
Secrets equal? True
```

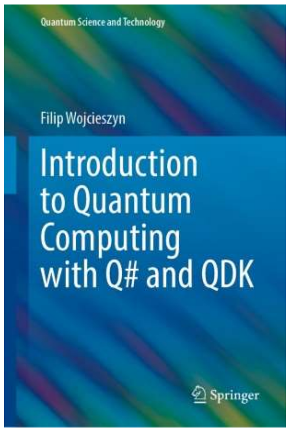
SOURCE CODE

I hope you find these examples useful - all the source code used in this article can be found on Github.

About

Hi! I'm **Filip W.**, a cloud architect from Zürich CH. I like Toronto Maple Leafs CA, Rancid and quantum computing. Oh, and I love the Lowlands .

You can find me on Github and on Mastodon.


Quantum Science and Technology
Filip Wojcieszyn
Introduction to Quantum Computing with Q# and QDK
Springer

Microsoft® MVP Most Valuable Professional

Recent Posts

2023/05/12, QIR Runner for ARM64 Macs is now available

2023/04/26, Building GPT powered applications with Azure OpenAI Service

2023/03/24, Post-quantum token signing with Dilithium using Duende Identity Server

2023/03/02, Beware of the default ASP.NET Core Identity settings

2023/02/21, Post quantum cryptography in .NET

📚 Full archive

Categories

ai (1)

apache-cordova (1)

asp.net-5 (17)

asp.net-core (44)

asp.net-mvc (35)

asp.net-mvc-6 (7)

asp.net-vnext (6)

asp.net-web-api (96)

astronomy (1)

azure (10)

azure-service-bus (1)

benchmark-dotnet (1)

bing-maps (1)

blazor (1)

c-plus (2)

c-sharp (152)

csharp-10 (2)

dnx (3)

dotnet-cli (2)

dotnet-script (10)

duende (2)

editorconfig (1)

entity-framework (2)

espn-api (2)

events (1)

fsharp (1)

git (1)

glimpse (1)

html5 (4)

identity-server (2)

iis (2)

il (1)

intro-to-qc (19)

javascript (9)

jquery (4)

jquery-mobile-metro (1)

katana (2)

kindle (1)

knockout.js (8)

last.fm-api (2)

linq (1)

mac (2)

mathematica (1)

msbuild (3)

mvc-core (3)

nancy (2)

native (1)

net (140)

net-5 (3)

net-6 (5)

net-7 (7)

net-core (49)

net-sdk (2)

ninject (2)

odata (4)

oidc (2)

omnisharp (13)

openai (1)

osx (2)

owin (5)

php (1)

q-sharp (30)

qir (2)

quantum-computing (32)

roslyn (30)

scriptcs (11)

scripting (9)

security (5)

servicestack (2)

signalr (7)

swift (2)

testing (5)

twitter-boostrap (1)

typescript (1)

visual-studio (4)

visual-studio-code (11)

wasi (2)

wasm (2)

windows-phone-7 (1)

wordpress (1)

wpf (2)

RSS Feed