Bouncy Castle

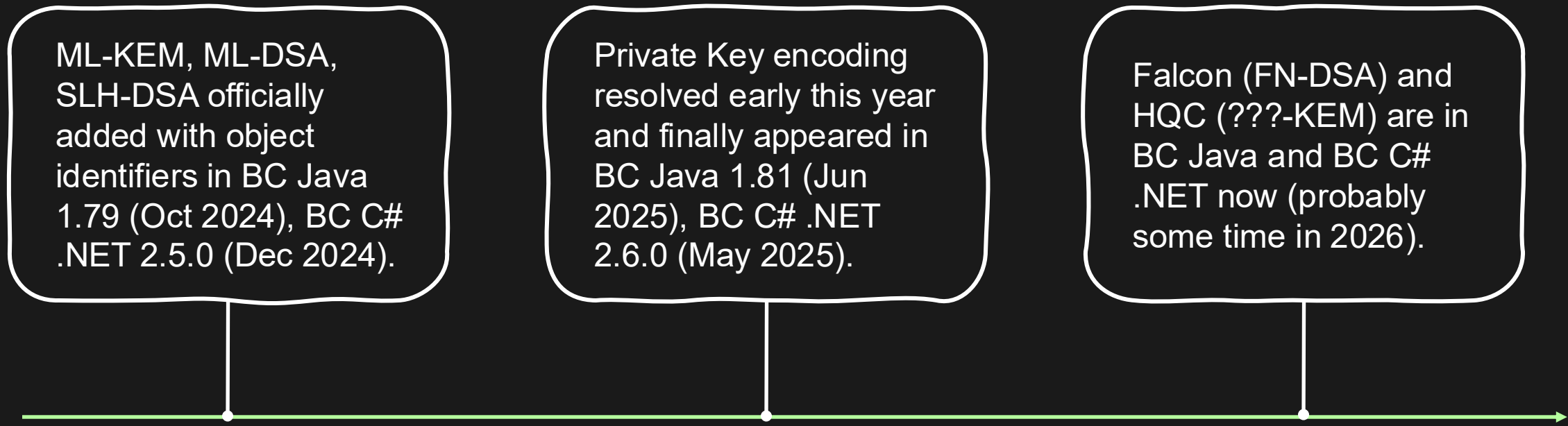
with Keyfactor

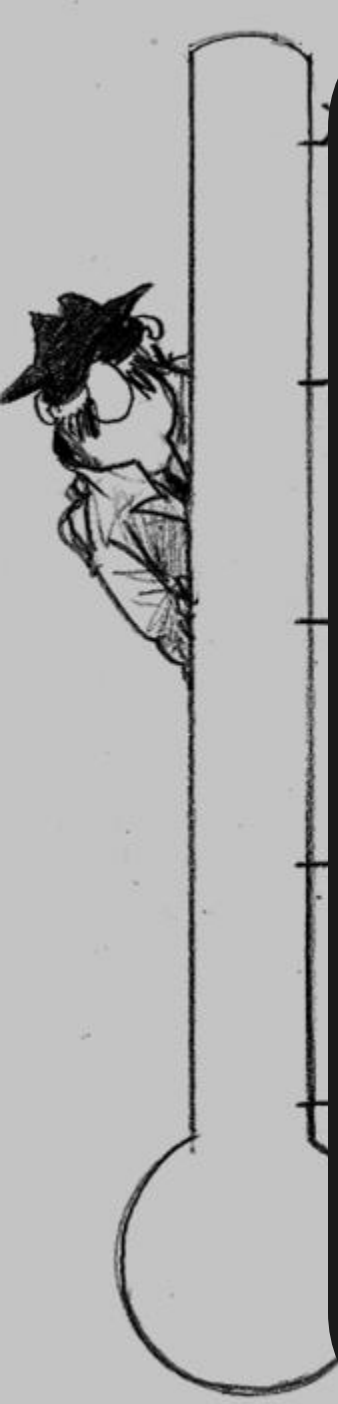
Bouncy Castle and a Post-Quantum future

API Changes, Protocol Issues, and Performance

David Hook, Founder,
Legion of the Bouncy Castle

Basic Timeline





So, what's the fuss?

- Key sizes, signature sizes, and key transport packets are much bigger than what we are used to (KEM packets 6400 to 12544 bits, Signatures 5728 to 398848 bits, keys from 256 bits to 20736 bits)
- There are three ways to store an ML-KEM/ML-DSA private key
- KEMs have one foot in the key transport club and one foot in the key agreement club. Oracle announced a new KEM API in Java 21 to allow proper support for them (BC support also through Cipher and KeyGenerator).
- There are currently no KEMs suitable for multi-party agreement.
- ML-KEM has been shown to be “efficient”, but ML-DSA and SLH-DSA do consume more CPU and memory resources. Bandwidth is another consideration.
- SLH-DSA needs to scan the message twice – can create issues which need to be solved by “pre-hash” rather than pure form.
- Pre-hash signature forms (using SHA2-512) exist for both ML-DSA and SLH-DSA.
- ML-DSA has an “external-mu” form which makes pre-hash (mostly) unnecessary
- There's been widespread interest in hybrid modes – using a PQC algorithm and a classical algorithm side by side (partially due to security worries, partially as a migration strategy)

ML Private Key Storage

ML-KEM and ML-DSA private keys can be stored 3 different ways:

- Seed-only
- Expanded-only
- Both the above

When "both the above" is encountered it's recommended to check the seed-only against the expanded-only form and then just run with the seed-only one.

For ML-DSA

```
KeyPairGenerator kpGen = KeyPairGenerator.getInstance("ML-DSA", "BC");

kpGen.initialize(MLDSAParameterSpec.ml_dsa_65, new SecureRandom());

KeyPair kp = kpGen.generateKeyPair();

MLDSAPrivateKey bothKey = (MLDSAPrivateKey)kp.getPrivate();  // full DER encoding 4098 bytes
PrivateKey seedOnlyKey = bothKey.getPrivateKey(true);        // full DER encoding 54 bytes!
```

KEM Based Key Transport (Pre-Java 21)

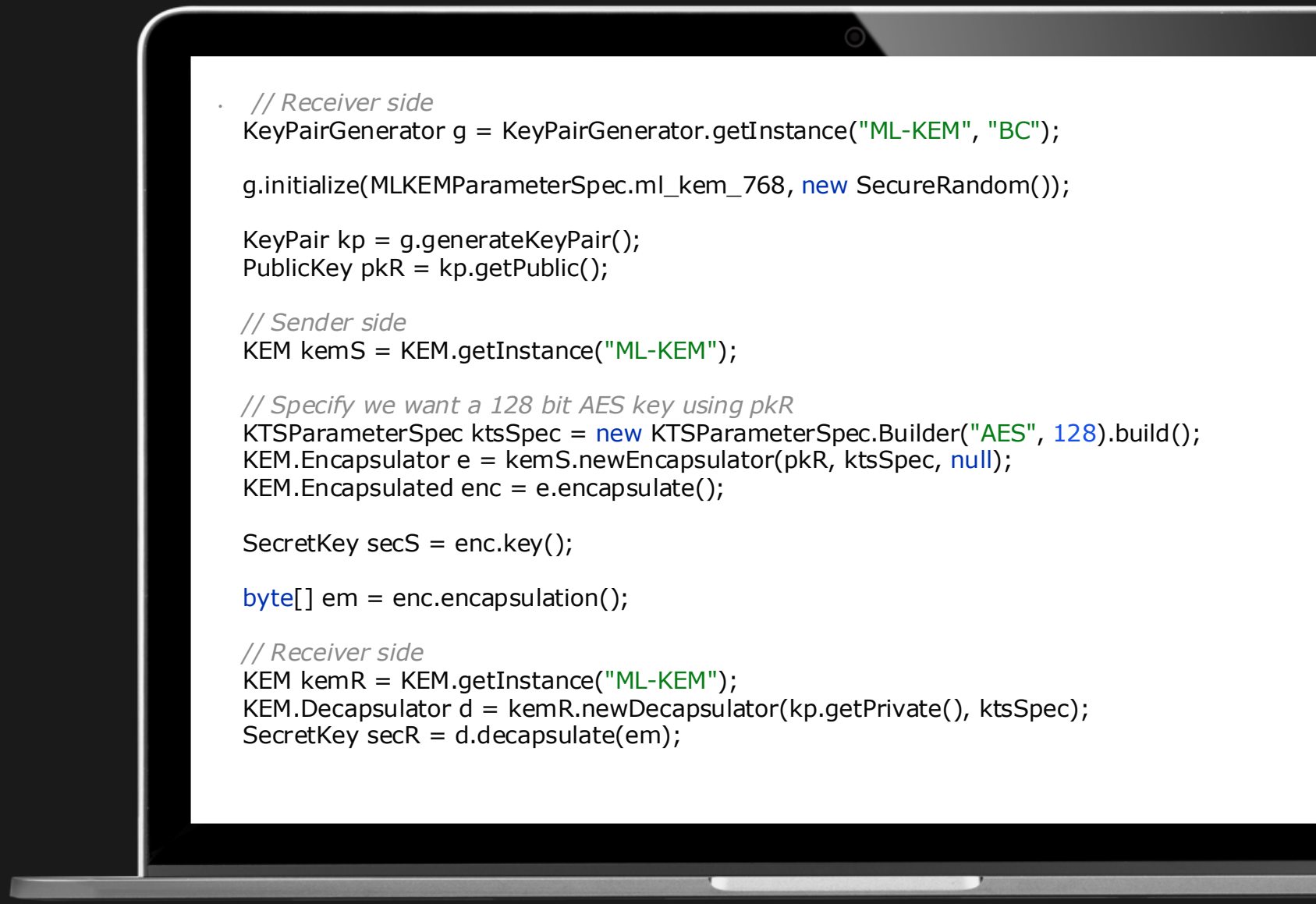
```
        // define our key to be wrapped
byte[] aesKey = Hex.decode("000102030405060708090a0b0c0d0e0f");

SecretKey key = new SecretKeySpec(aesKey, "AES");

// generate an ML-KEM-768 key pair
KeyPairGenerator kpg = KeyPairGenerator.getInstance("MLKEM", "BC");

kpg.initialize(MLKEMParameterSpec.ml_kem_768, new SecureRandom());

KeyPair kp = kpg.generateKeyPair();

// specify we want to use 256 bit AES-KWP for wrapping our key
KTSParameterSpec ktsParameterSpec = new KTSParameterSpec.Builder("AES-KWP", 256).build();

// set up the wrapping cipher
Cipher w1 = Cipher.getInstance("MLKEM", "BC");

w1.init(Cipher.WRAP_MODE, kp.getPublic(), ktsParameterSpec);

// wrap the key
byte[] data = w1.wrap(key);

// set up the unwrapping cipher
Cipher w2 = Cipher.getInstance("MLKEM", "BC");

w2.init(Cipher.UNWRAP_MODE, kp.getPrivate(), ktsParameterSpec);

// unwrap the encrypted key
Key k = w2.unwrap(data, "AES", Cipher.SECRET_KEY);
```

Bouncy Castle
with Keyfactor

Key Generation for AES (Pre-Java 21)

```
// generate ML-KEM-512 key pair.
KeyPairGenerator kpg = KeyPairGenerator.getInstance("ML-KEM", "BC");

kpg.initialize(MLKEMParameterSpec.ml_kem_512, new SecureRandom());

KeyPair kp = kpg.generateKeyPair();

// Create the KeyGenerator - there are specific specs for creating encapsulations
// (KEMGenerateSpec) and for extract a secret key from an encapsulation using the
// private key (KEMExtractSpec)
KeyGenerator keyGen = KeyGenerator.getInstance("ML-KEM", "BC");

// initialise for creating an encapsulation and shared secret.
keyGen.init(new KEMGenerateSpec(kp.getPublic(), "AES", 128), new SecureRandom());

// SecretKeyWithEncapsulation is the class to use as the secret key, it has additional
// methods on it for recovering the encapsulation as well.
SecretKeyWithEncapsulation secEnc1 = (SecretKeyWithEncapsulation)keyGen.generateKey();

keyGen.init(new KEMExtractSpec(kp.getPrivate(), secEnc1.getEncapsulation(), "AES", 128));

// initialise for extracting the shared secret from the encapsulation.
SecretKeyWithEncapsulation secEnc2 = (SecretKeyWithEncapsulation)keyGen.generateKey();
```

Java 21 – KEM API



```
// Receiver side
KeyPairGenerator g = KeyPairGenerator.getInstance("ML-KEM", "BC");

g.initialize(MLKEMParameterSpec.ml_kem_768, new SecureRandom());

KeyPair kp = g.generateKeyPair();
PublicKey pkR = kp.getPublic();

// Sender side
KEM kemS = KEM.getInstance("ML-KEM");

// Specify we want a 128 bit AES key using pkR
KTSParameterSpec ktsSpec = new KTSParameterSpec.Builder("AES", 128).build();
KEM.Encapsulator e = kemS.newEncapsulator(pkR, ktsSpec, null);
KEM.Encapsulated enc = e.encapsulate();

SecretKey secS = enc.key();

byte[] em = enc.encapsulation();

// Receiver side
KEM kemR = KEM.getInstance("ML-KEM");
KEM.Decapsulator d = kemR.newDecapsulator(kp.getPrivate(), ktsSpec);
SecretKey secR = d.decapsulate(em);
```

ML-DSA External Mu

FIPS PUB 204, Page 25:

$\mu \leftarrow$ H(BytesToBits(tr) || M', 64) ▷ message representative that may optionally be computed in a different cryptographic module

After some discussion it was decided to allow this as an additional approach which would avoid the use of pre-hash for large messages.

For implementations this means offering two stage signature generation and verification:

- calculate μ from SHAKE256(public key), domain separator, context (if provided), and message
- calculate or verify signature based on μ value.

External Mu Calculation

Two approaches:

With Private Key

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("ML-DSA", "BC");

kpg.initialize(MLDSAParameterSpec.ml_dsa_65,  new SecureRandom());

KeyPair kp = kpg.generateKeyPair();

Signature sig = Signature.getInstance("ML-DSA-CALCULATE-MU", "BC");
sig.initSign(kp.getPrivate());
```

With Public Key

```
sig = Signature.getInstance("ML-DSA-CALCULATE-MU", "BC");

sig.initSign(new MLDSAProxyPrivateKey(kp.getPublic()));
```

Signing with External Mu

- The calculated Mu value becomes the message
- Two options: deterministic/non-deterministic

Deterministic

```
sig = Signature.getInstance("ML-DSA-EXTERNAL-MU", "BC");

sig.initSign(kp.getPrivate());

sig.update(mu, 0, mu.length);

byte[] s = sig.sign();
```

Non-Deterministic

```
sig = Signature.getInstance("ML-DSA-EXTERNAL-MU", "BC");

sig.initSign(kp.getPrivate(), new SecureRandom());

sig.update(mu, 0, mu.length);

byte[] s = sig.sign();
```

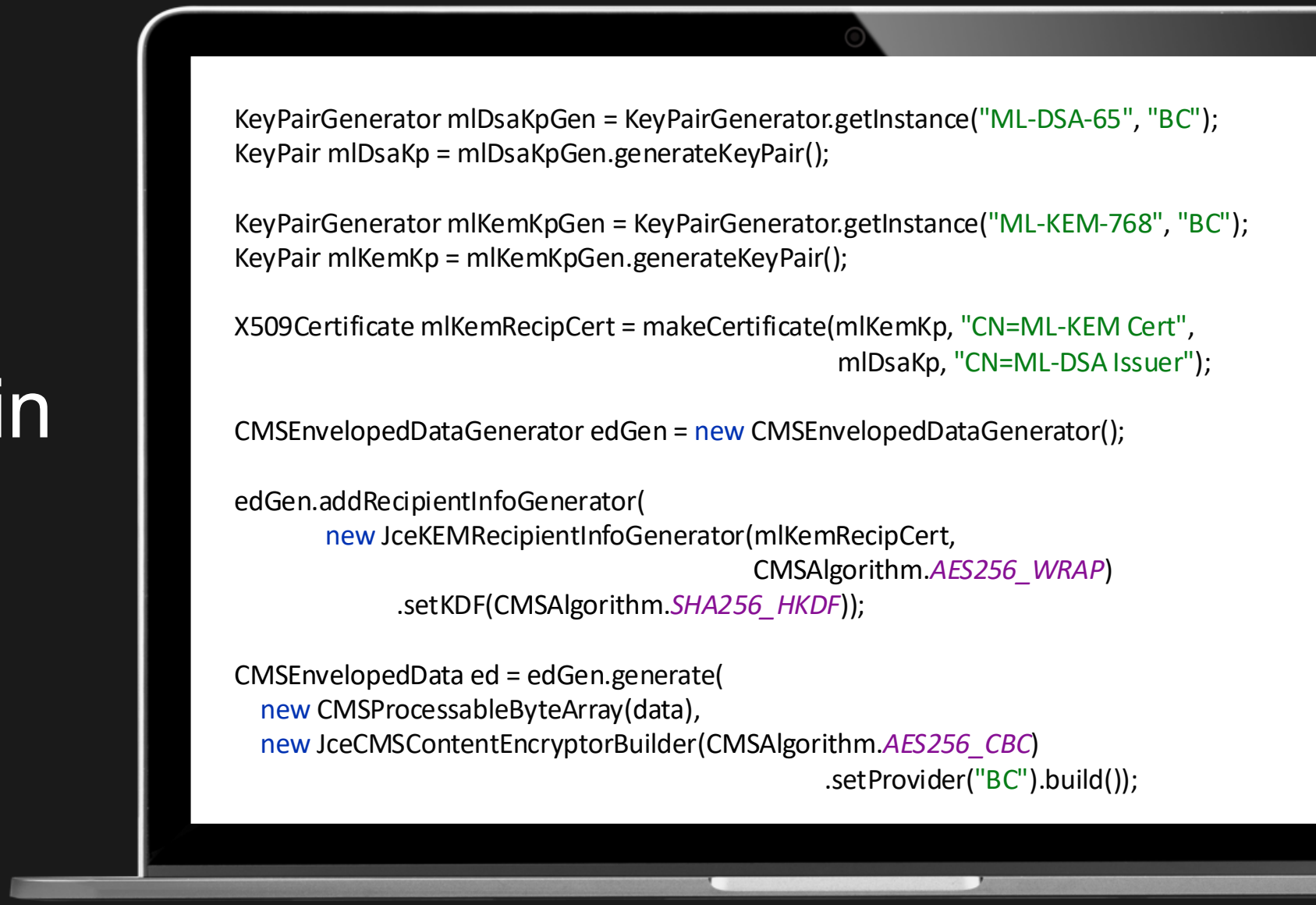
CMS

ML-DSA and SLH-DSA are fine. Payloads will be bigger due to signature and public key sizes.

KeyTransRecipientInfo, the traditional path for EnvelopedData recipients, will be replaced with KEMRecipientInfo.

KEMRecipientInfo also carries KDF information to allow a KEM generated secret to be turned into a symmetric key.

KEM EnvelopedData in CMS



```
KeyPairGenerator mlDsaKpGen = KeyPairGenerator.getInstance("ML-DSA-65", "BC");
KeyPair mlDsaKp = mlDsaKpGen.generateKeyPair();

KeyPairGenerator mlKemKpGen = KeyPairGenerator.getInstance("ML-KEM-768", "BC");
KeyPair mlKemKp = mlKemKpGen.generateKeyPair();

X509Certificate mlKemRecipCert = makeCertificate(mlKemKp, "CN=ML-KEM Cert",
                                                 mlDsaKp, "CN=ML-DSA Issuer");

CMSEnvelopedDataGenerator edGen = new CMSEnvelopedDataGenerator();

edGen.addRecipientInfoGenerator(
        new JceKEMRecipientInfoGenerator(mlKemRecipCert,
                                         CMSAlgorithm.AES256_WRAP)
                .setKDF(CMSAlgorithm.SHA256_HKDF));

CMSEnvelopedData ed = edGen.generate(
   new CMSProcessableByteArray(data),
   new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_CBC)
                                         .setProvider("BC").build());
```

Bouncy Castle
with Keyfactor

Decrypting KEM EnvelopedData in CMS

The decryption process looks almost exactly like it does for RSA

Alternately a subjectKeyID can be used on both ends to identify the recipient KEM key in the same way as for RSA.

```
CMSEnvelopedData ed = … // from previous slide

RecipientInformationStore recipients = ed.getRecipientInfos();

KEMRecipientInformation recipient = (KEMRecipientInformation)recipients.get(
                                        new JceKEMRecipientId(mlKemRecipCert));

CMSTypedStream contentStream = recipient.getContentStream(
   new JceKEMEnvelopedRecipient(mlKemKp.getPrivate()).setProvider("BC"));

byte[] decData = Streams.readAll(contentStream.getContentStream());
```

What does this mean for PKI?

KEMs cannot be used to generate signatures.

- ✗ if using PKCS#10 need an associated signature.
- ✗ if using CRMF/CMP “certEncr” is approach if Proof-of-Possession is required.

There are now four proposals for certificate extensions and signature types for assisting with migration or allowing side by side use of classical and post-quantum algorithms:

- Chimera (X.509 dual key)
- Chameleon (IETF draft – also allows for second key and signature)
- Composite (IETF draft – combines two signature algorithms)
- Related Certificates (RFC 9763)

Bouncy Castle
with Keyfactor

KEM Certification Request Generation - PKCS#10

```
KeyPairGenerator dilKpGen = KeyPairGenerator.getInstance("ML-DSA", "BC");

dilKpGen.initialize(MLDSAParameterSpec.ml_dsa_65);

KeyPair dilKp = dilKpGen.generateKeyPair();

X509CertificateHolder sigCert = makeV3Certificate("CN=ML-KEM Client", dilKp);

KeyPairGenerator kemKpGen = KeyPairGenerator.getInstance("ML-KEM", "BC");

kemKpGen.initialize(MLKEMParameterSpec.ml_kem_768);

KeyPair kemKp = kemKpGen.generateKeyPair();

PKCS10CertificationRequestBuilder pkcs10Builder = new JcaPKCS10CertificationRequestBuilder(
                        new X500Name("CN=ML-KEM Client"), kemKp.getPublic());

pkcs10Builder.addAttribute(X509AttributeIdentifiers.id_at_privateKeyStatement,
                    new PrivateKeyStatement(sigCert.toASN1Structure()));

PKCS10CertificationRequest request = pkcs10Builder.build(
                new JcaContentSignerBuilder("ML-DSA").setProvider("BC").build(dilKp.getPrivate()));
```

Bouncy Castle
with Keyfactor

KEM Certification Request Generation – CRMF

```
JcaCertificateRequestMessageBuilder certReqBuild =
                                        new JcaCertificateRequestMessageBuilder(BigIntegers.ONE);

certReqBuild
   .setPublicKey(kemKp.getPublic())
   .setSubject(X500Name.getInstance(sender.getName()))
   .setProofOfPossessionSubsequentMessage(SubsequentMessage.encrCert)
   .addControl(new JcaPKIArchiveControlBuilder(
         ((MLKEMPrivateKey)kemKp.getPrivate()).getPrivateKey(true),    // seed only private key
         new X500Principal("CN=ML-KEM Subject"))
            .addRecipientGenerator(new JceKEMRecipientInfoGenerator(caEncCert,
                                                   CMSAlgorithm.AES256_WRAP).setProvider("BC"))
            .build(new JceCMSContentEncryptorBuilder(CMSAlgorithm.AES256_GCM)
                                                                .setProvider("BC").build()));

CertificateReqMessagesBuilder certReqMsgsBldr = new CertificateReqMessagesBuilder();

certReqMsgsBldr.addRequest(certReqBuild.build());

MacCalculator senderMacCalculator = new JcePBMac1CalculatorBuilder("HmacSHA256", 256)
                                                    .setProvider("BC").build(senderMacPassword);

ProtectedPKIMessage message = new ProtectedPKIMessageBuilder(sender, recipient)
   .setBody(PKIBody.TYPE_INIT_REQ, certReqMsgsBldr.build())
   .build(senderMacCalculator);
```

Bouncy Castle
with Keyfactor

Hybrid Certificate Formats

Bouncy Castle
with Keyfactor

Chimera

Originally published in X.509 2019

Defines 3 new extensions:

- SubjectAltPublicKeyInfo,
- AltSignatureAlgorithm,
- AltSignatureValue

Extensions marked as non-critical, allowing implementations to ignore if cannot support

Requires different verification approach to verify certificate against altSignatureValue from regular certificate

X.509 Chimera Certificate Generation

```
ContentSigner ecSigner = new
JcaContentSignerBuilder("SHA256withECDSA").setProvider("BC").build(ecPrivKey);
ContentSigner mldsaSigner = new JcaContentSignerBuilder("ML-DSA-
65").setProvider("BC").build(mldsaPrivKey);

X509v3CertificateBuilder certGen = new JcaX509v3CertificateBuilder(
   new X500Name("CN=Chimera Example"),
   BigInteger.valueOf(1), new Date(System.currentTimeMillis() - 50000),
    new Date(System.currentTimeMillis() + SIX_MONTHS),
   new X500Name("CN=Chimera Example"), ecPubKey)
   .addExtension(Extension.basicConstraints, true, new BasicConstraints(false))
   .addExtension(Extension.subjectAltPublicKeyInfo, false,
                                    SubjectAltPublicKeyInfo.getInstance(mldsaPubKey.getEncoded()));

X509Certificate cert = new JcaX509CertificateConverter().setProvider("BC")
                                                .getCertificate(certGen.build(ecSigner, false, mldsaSigner));

X509CertificateHolder certHolder = new JcaX509CertificateHolder(cert);

boolean altSigValid = certHolder.isAlternativeSignatureValid(
                        new JcaContentVerifierProviderBuilder().setProvider("BC")
                                                               .build(mldsaPubKey));
```

Bouncy Castle
with Keyfactor

Chameleon

IETF Draft: draft-bonnell-lamps-chameleon-cert (version 6)

Defines 1 new extension:

- DeltaCertificate Descriptor

Contains enough information to construct a new certificate from fields in the parent certificate and the extension

Extensions marked as non-critical, allowing implementations to ignore if cannot support

Does not require different verification algorithm for alternative key as allows for creation of a standalone certificate

Chameleon Certificate Generation

```
// Base ML-DSA certificate for delta extension
X509v3CertificateBuilder deltaBldr = new X509v3CertificateBuilder(
   new X500Name("CN=Chameleon CA 2"),
   BigInteger.valueOf(System.currentTimeMillis()), notBefore, notAfter,
   new X500Name("CN=Delta Subject"),
   SubjectPublicKeyInfo.getInstance(mldsaPubKey.getEncoded()));

deltaBldr.addExtension(Extension.basicConstraints, true, new BasicConstraints(false));

X509CertificateHolder deltaCert = deltaBldr.build(mldsaSigner);

// Build the Chameleon Certificate
X509v3CertificateBuilder bldr = new X509v3CertificateBuilder(
   new X500Name("CN=Chameleon CA 1"),
   BigInteger.valueOf(System.currentTimeMillis()), notBefore, notAfter,
   new X500Name("CN=Delta Subject"),
   SubjectPublicKeyInfo.getInstance(ecPubKey.getEncoded()));

// Create an extension containing a template for the DeltaCertificateDescriptor
Extension deltaExt = DeltaCertificateTool.makeDeltaCertificateExtension(false, deltaCert);

bldr.addExtension(deltaExt);

X509CertificateHolder chameleonCert = bldr.build(ecSigner);

// extract the DeltaCertificateDescriptor and rebuild the cert
DeltaCertificateDescriptor deltaCertDesc = DeltaCertificateDescriptor.fromExtensions(chameleonCert.getExtensions());

X509CertificateHolder exDeltaCert = DeltaCertificateTool.extractDeltaCertificate(chameleonCert);
```

Hybrid Key Agreement

SP 800-56C defines approaches for using 2 shared secrets to feed into a KDF.

Bouncy Castle provides the HybridValueParameterSpec which uses a concatenation combiner (T value second) inside a key agreement operation.

For certification enthusiasts:

- only one secret needs to be generated using an approved-mode technique
- it doesn't matter what order the approved-mode and not-approved-mode secret are mixed in

Hybrid Key Agreement Example

```
// ecKp1, ecKp2 are 2 P-521 key pairs, kemKp is a KEM key-pair belonging to one party
KeyGenerator keyGen = KeyGenerator.getInstance("ML-KEM", "BC");

// Generate shared secret and encapsulation - no KDF as one applied in agreement
keyGen.init(new KEMGenerateSpec.Builder(kemKp.getPublic(), "T", 256).withNoKdf().build(), new SecureRandom());

SecretKeyWithEncapsulation secEnc1 = (SecretKeyWithEncapsulation)keyGen.generateKey();

KeyAgreement agree1 = KeyAgreement.getInstance("ECCDHwithSHA256CKDF", "BC");

// Perform key agreement using the shared secret as a T value
agree1.init(ecKp1.getPrivate(), new HybridValueParameterSpec(secEnc1.getEncoded(), new UserKeyingMaterialSpec(ukm)));

agree1.doPhase(ecKp2.getPublic(), true);

// Generate originator agreed AES key.
SecretKey k1 = agree1.generateSecret("AES[256]");

// Generate shared secret from encapsulation - no KDF as one applied in agreement
keyGen.init(new KEMExtractSpec.Builder(kemKp.getPrivate(), secEnc1.getEncapsulation(), "T", 256).withNoKdf().build());

SecretKeyWithEncapsulation secEnc2 = (SecretKeyWithEncapsulation)keyGen.generateKey();

KeyAgreement agree2 = KeyAgreement.getInstance("ECCDHwithSHA256CKDF", "BC");

// Perform key agreement using the shared secret as a T value
agree2.init(ecKp2.getPrivate(), new HybridValueParameterSpec(secEnc2.getEncoded(), new UserKeyingMaterialSpec(ukm)));

agree2.doPhase(ecKp1.getPublic(), true);

// Generate recipient agreed AES key.
SecretKey k2 = agree2.generateSecret("AES[256]");
```

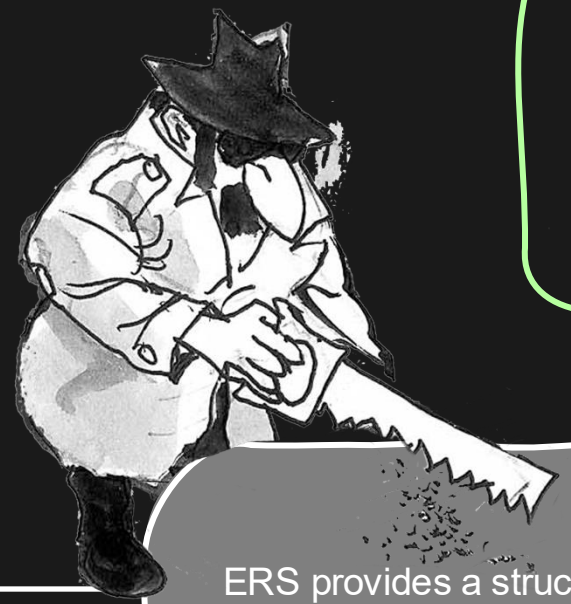
Time Stamping

Producing a PQC time stamp is pretty much like producing a classical one – just different algorithm.

The issue is more about what to do about all the classical ones, particular those that might need to be extended.

- RFC 3126 (now RFC 5126) added an archive time stamp that could be stored as an attribute in Cryptographic Message Syntax messages, or S/MIME messages.
- Evidence Record Syntax (ERS) (RFC 4998, RFC 6283) builds on these and provides for a more sophisticated structure, the ArchiveTimeStamp based on a Merkle tree.
- A mechanism is also provided for refreshing an ArchiveTimeStamp to allow the original time stamp to be safely preserved in the future.

ERS – Time Stamp maintenance



Two situations can arise that might require time stamp renewal:

- an issue with the hash algorithm
- an issue with the signature

ERS provides a structure ArchiveTimeStampChain which is simply a sequence of ArchiveTimeStamp structures all using the same hash algorithm.

Another structure, the ArchiveTimeStampSequence, is a sequence of ArchiveTimeStampChain structures.

A new ArchiveTimeStampChain is created each time a new hash algorithm is used.

Creating an ArchiveTimeStamp

In its simplest form an ArchiveTimeStamp just contains a classic time stamp representing a time stamp on a single document.

In the more general case the structure also includes a PartialHashTree which is a structure provides data that can be used to build a Merkle tree.

It is the root node of the Merkle tree which is then time stamped using the usual approach.

Basic Look at Evidence Record

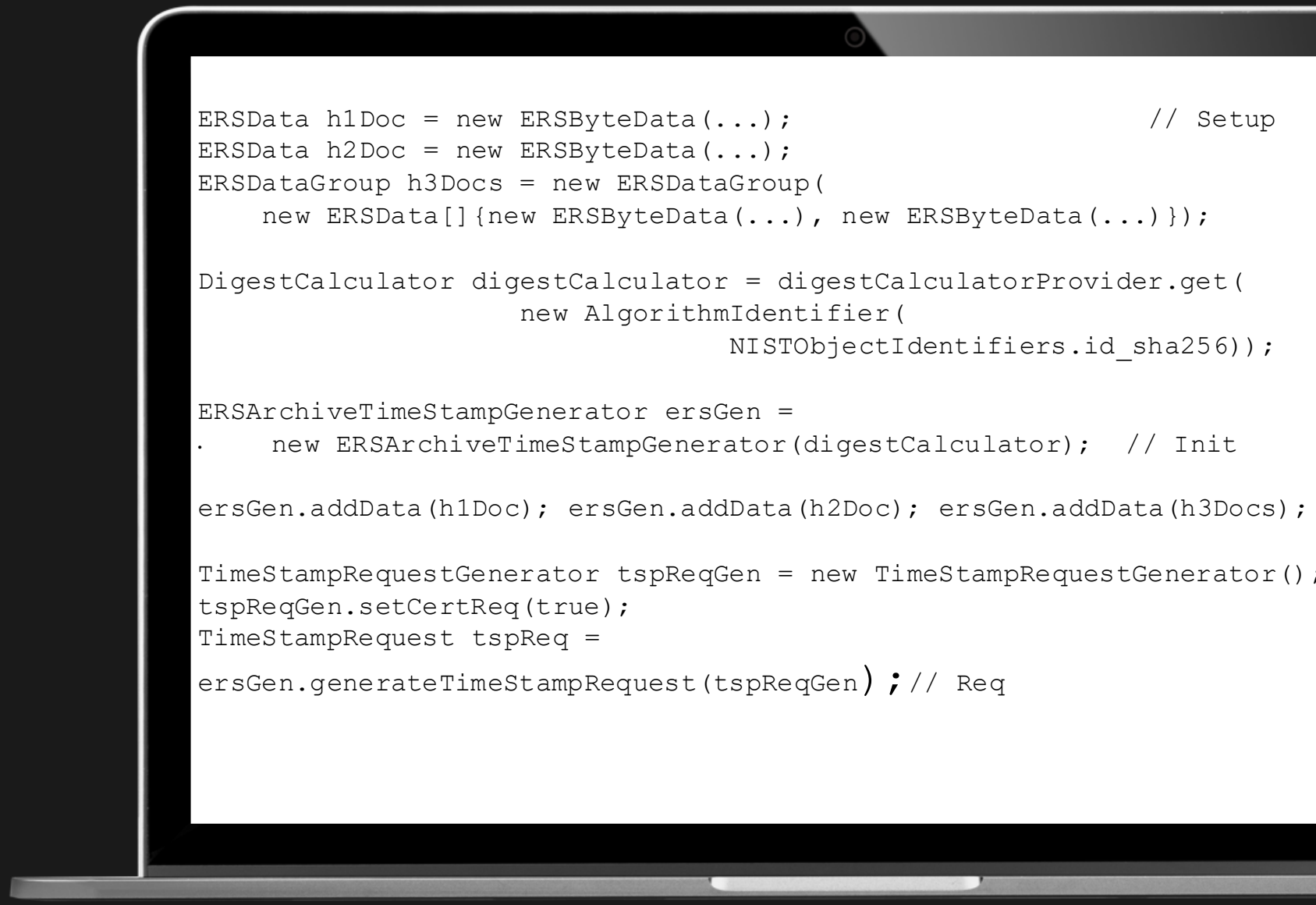
ERS talks about building trees out of Data Objects and Data Object Groups.

In Bouncy Castle we do this by using:

- ERSData which specifies a simple leaf node in the tree
- ERSGroupData which specifies a group of leaf nodes that should have a common parent.

The ERSArchiveTimeStampGenerator is then used to bring the data objects and data object groups together.

Archive Time Stamp Setup



```
ERSData h1Doc = new ERSByteData(...);                          // Setup
ERSData h2Doc = new ERSByteData(...);
ERSDataGroup h3Docs = new ERSDataGroup(
    new ERSData[]{new ERSByteData(...), new ERSByteData(...)});

DigestCalculator digestCalculator = digestCalculatorProvider.get(
                    new AlgorithmIdentifier(
                                    NISTObjectIdentifiers.id_sha256));

ERSArchiveTimeStampGenerator ersGen =
.    new ERSArchiveTimeStampGenerator(digestCalculator);  // Init

ersGen.addData(h1Doc); ersGen.addData(h2Doc); ersGen.addData(h3Docs);

TimeStampRequestGenerator tspReqGen = new TimeStampRequestGenerator();
tspReqGen.setCertReq(true);
TimeStampRequest tspReq =
ersGen.generateTimeStampRequest(tspReqGen);// Req
```

Bouncy Castle
with Keyfactor

Archive Time Stamp Finalization

Once the time stamp response comes back, the completed ArchiveTimeStamp is made using it:

```
TimeStampResponse tsResp = ...

ERSArchiveTimeStamp ats =
                        ersGen.generateArchiveTimeStamp(tsResp);
```

The ERSArchiveTimeStamp is the Bouncy Castle equivalent to RFC4998's ArchiveTimeStamp.

Support for using LMS, ML-DSA, and SLH-DSA for ArchiveTimeStamp and TimeStamp generation now available in Bouncy Castle.

TLS

BCTLS, as of version 1.81/2.6.1, supports three named groups for ML-KEM described in draft-ietf-tls-mlkem

BCTLS, as of version 1.81/2.6.1, supports the hybrid algorithms X25519MLKEM768, SecP256r1MLKEM768, and SecP384r1MLKEM102 defined in draft-ietf-tls-ecdhe-mlkem

Interop

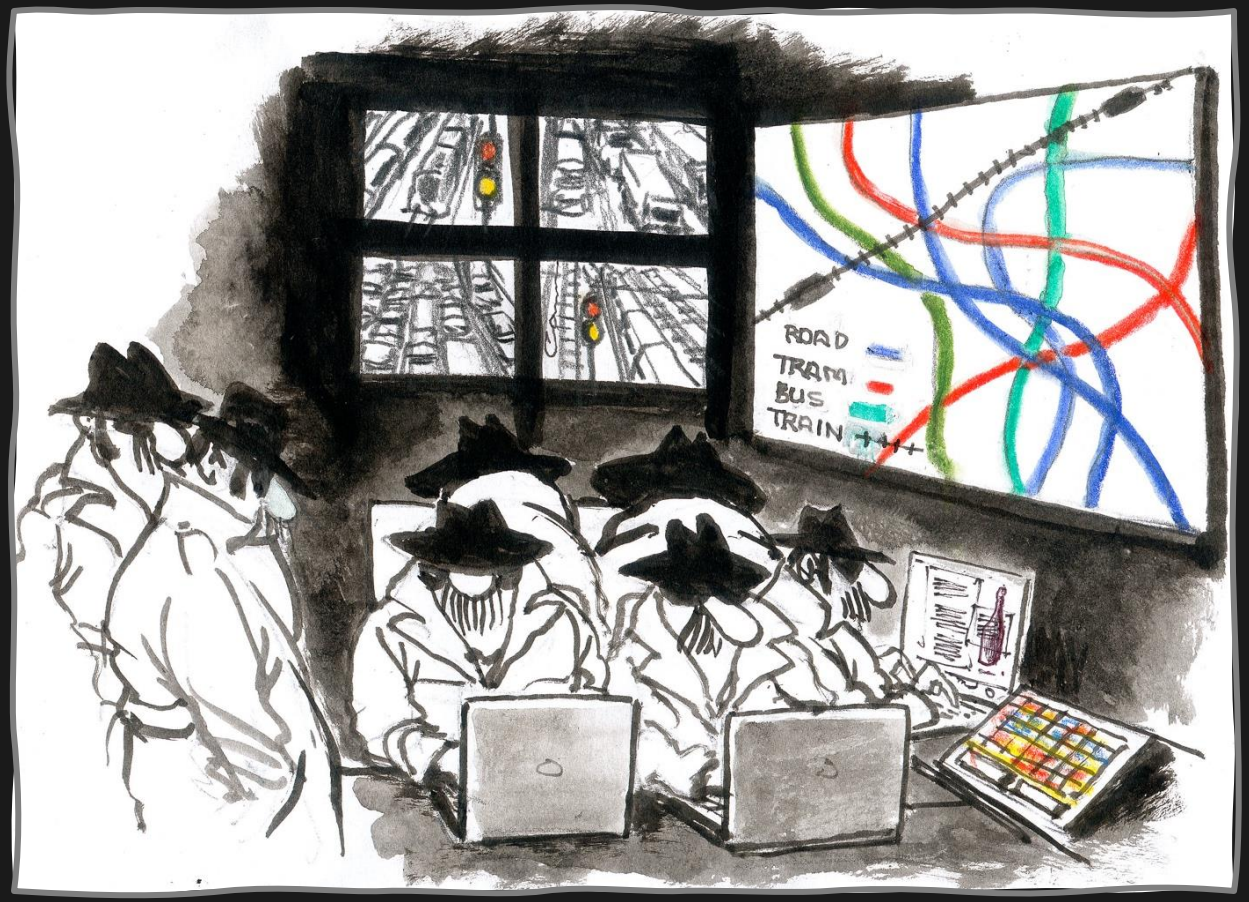
Surprisingly few interop issues with X.509 (thanks IETF PQC Hackathon)

Private keys have been a real problem

Some issues with Falcon (detached vs un-detached signature encoding)

For the most part there have been few issues with TLS as well (straight ML-KEM and Hybrid ML-KEM).

Performace



- For pure Java/C#, it is still early days.
- Section 7 of the Preamble in FIPS PUB 203, and Section 8 of the Preambles in FIPS PUB 204/205 state that mathematical equivalence can be used to justify changes to the algorithms as described.
- This is interesting as in some cases pseudo randomness is being generated using hashing – there have already been some proposals to replace these with actual RNGs.
- For ML-DSA and ML-KEM we have made attempts to use SHA-3 natively, but the cost of crossing the JVM boundary is higher than the performance improvement.
- Native wrapped ML-KEM offers a throughput improvement between 7 and 9 times that obtainable with pure Java. ML-DSA about 3 times.

Additional Resources

PQC Almanac

https://downloads.bouncycastle.org/java/docs/PQC-Almanac.pdf

PQC Almanac Source Code

https://downloads.bouncycastle.org/java/docs/PQC-Almanac-Examples.zip

Replace Java with C # in the link for C # versions.

Bouncy Castle
with Keyfactor

Thanks for Listening

Bouncy Castle
with Keyfactor