Practical Cryptography

with Tink

illuminated

Neil Madden

Author *API Security in Action*

Found “Psychic Signatures” vulnerability in Java ECDSA

AppSec and applied crypto specialist

Contributor to OAuth and JOSE working groups

illuminated

Tink

Developed by Google

Java, Go, C++, Obj-C, Python

Designed by cryptographers

Easy to use securely

AEAD

Authenticated Encryption (with Associated Data)

Confidentiality + Integrity

Data Origin Authentication

Encrypting a file

```
var cipher =
    Cipher.getInstance("AES/GCM/NoPadding");
cipher.init(Cipher.ENCRYPT_MODE, key);
var iv = cipher.getIV();
cipher.updateAAD(filename.getBytes());
var ciphertext = cipher.doFinal(data);

try (var out =
      new FileOutputStream(filename)) {
    out.write(iv);
    out.write(ciphertext);
}
```

illuminated

Decrypting a file

```
var cipher =
    Cipher.getInstance("AES/GCM/NoPadding");

try (var in = new FileInputStream(filename)) {
    var iv = in.readNBytes(12);
    var ciphertext = in.readAllBytes();

    cipher.init(Cipher.DECRYPT_MODE, key,
        new GCMParameterSpec(128, iv));
    cipher.updateAAD(filename.getBytes());
    return cipher.doFinal(ciphertext);
}
```

illuminated

Old-style crypto libraries

- Very general abstractions
- Lots of algorithm choices
- Expert user assumption
- Easy to misuse

rawpixel.com / US Department of Defense (Public Domain)

Modern cryptography libraries

- Tink
- NaCl (“salt”) / libsodium
- Sandwich
- Themis

Tink: encrypting a (small) file

```
Aead aead = keyset.getPrimitive(Aead.class);
try (var out = new FileOutputStream(filename)) {
    out.write(aead.encrypt(data, filename.getBytes());
}
```

Decrypting

```
Aead aead = keyset.getPrimitive(Aead.class);
try (var in = new FileInputStream(filename)) {
    return aead.decrypt(in.readAllBytes(),
        filename.getBytes());
}
```

Tink design principles

- Each “primitive” has its own interface like `Aead`
- All implementations of that interface achieve the same *security goal*
- Different choices may have different performance profiles, but all are secure
- Easy to misuse options are hidden from users

Supported AEAD algorithms

Algorithm	Pros	Cons
AES-GCM	Fast on most CPUs	Max 2^{32} messages, fragile
AES-CTR+HMAC	Simple, robust	Slower
AES-EAX	Simple, robust	Rarely used
AES-GCM-SIV	Fast, robust	Rarely used, max 2^{32} messages
ChaCha20-Poly1305	Fast in software	Max 2^{32} messages
XChaCha20-Poly1305	Fast, 2^{80} messages(!)	Rarely used

illuminated

Principle:

Tink supports many algorithms, but they all implement the <u>same</u> security goal.

illuminated

> "Cryptography is a tool for turning lots of different problems into key management problems."

— Lea Kissner

illuminated

Key sets

and cryptographic agility

illuminated

Dangerous cryptographic agility?

It has been 245 days since the last alg:none JWT vulnerability.

An unauthenticated attacker could impersonate any user in SharePoint 2019 by using an alg:none JWT for OAuth authentication.

illuminated

“If you have to perform any cryptographic operation before [authenticating] a message you’ve received, it will somehow inevitably lead to doom.”

— Moxie Marlinspike

illuminated

Confused Deputies

{"alg":"HS256"}

RSA Public Key

illuminated

Key-driven cryptographic agility

O'REILLY

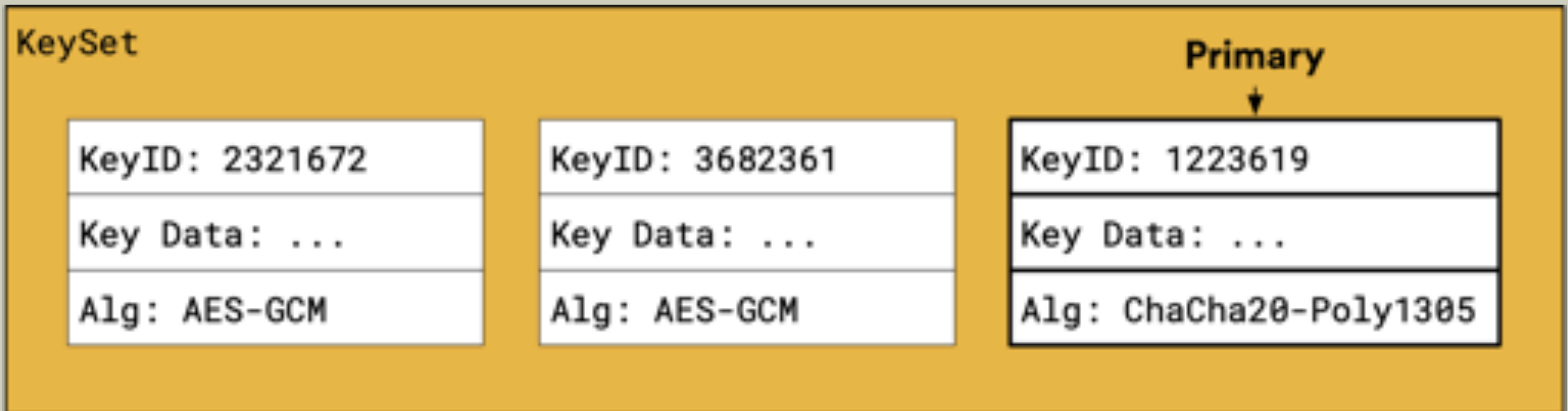
Software Engineering at Google

Lessons Learned from Programming Over Time

Curated by Titus Winters, Tom Manshreck & Hyrum Wright

illuminated

Keysets



Tink AEAD format

- By default, Tink prepends a Key ID and the IV to the ciphertext.
- Provides `RAW` variants that do not include Key ID

illuminated

Creating a keyset

```
tinkey create-keyset \
  --key-template AES256_GCM \
  --out-format json \
  --out keys.json
```

Warning: insecure unencrypted format

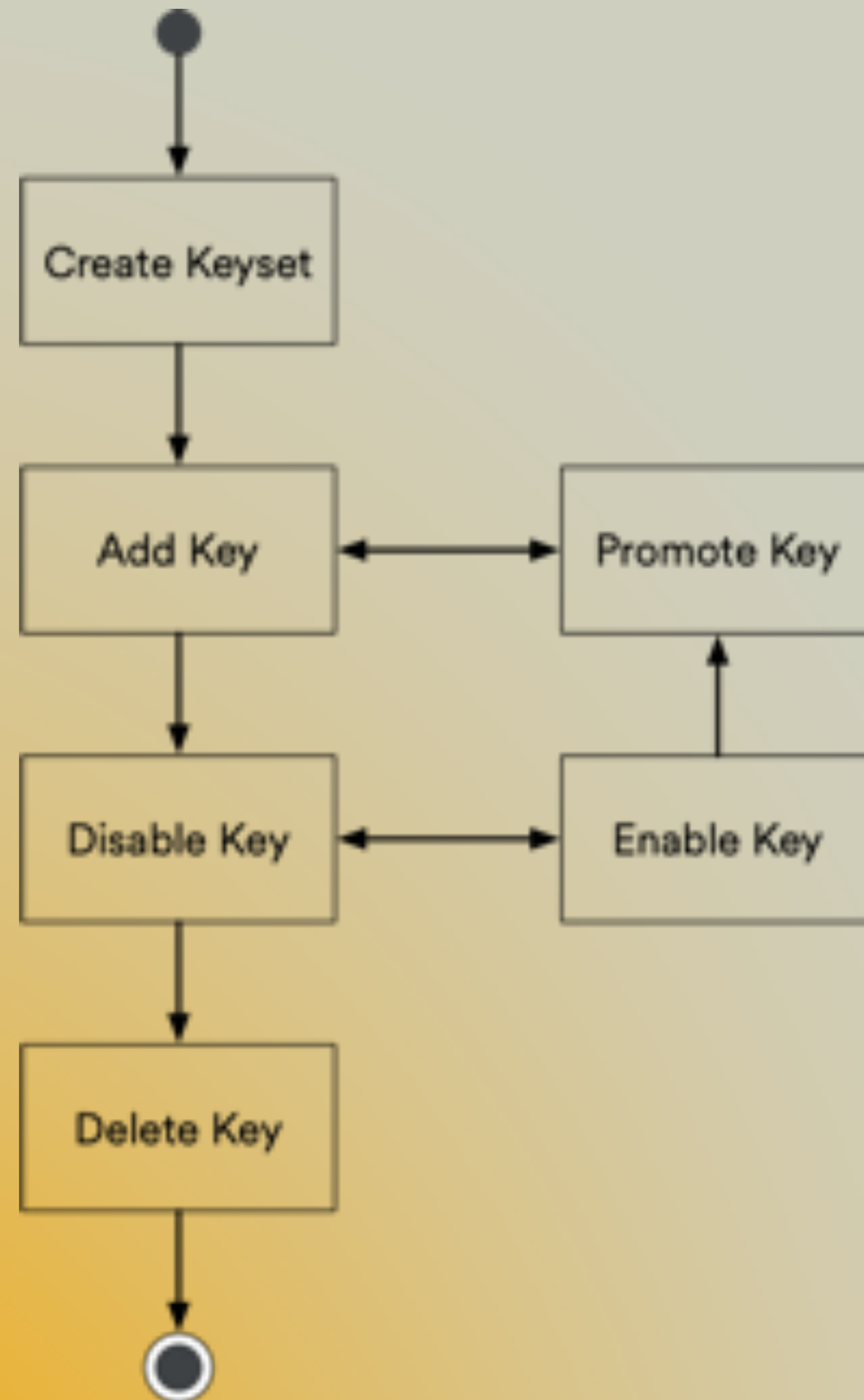
Using a keyset

```
var keysetData =
    Files.readString(
      Paths.get("keys.json"));

return
  TinkJsonProtoKeysetFormat.parseKeyset(
        keysetData,
        InsecureSecretKeyAccess.get());
```

illuminated

Key lifecycle



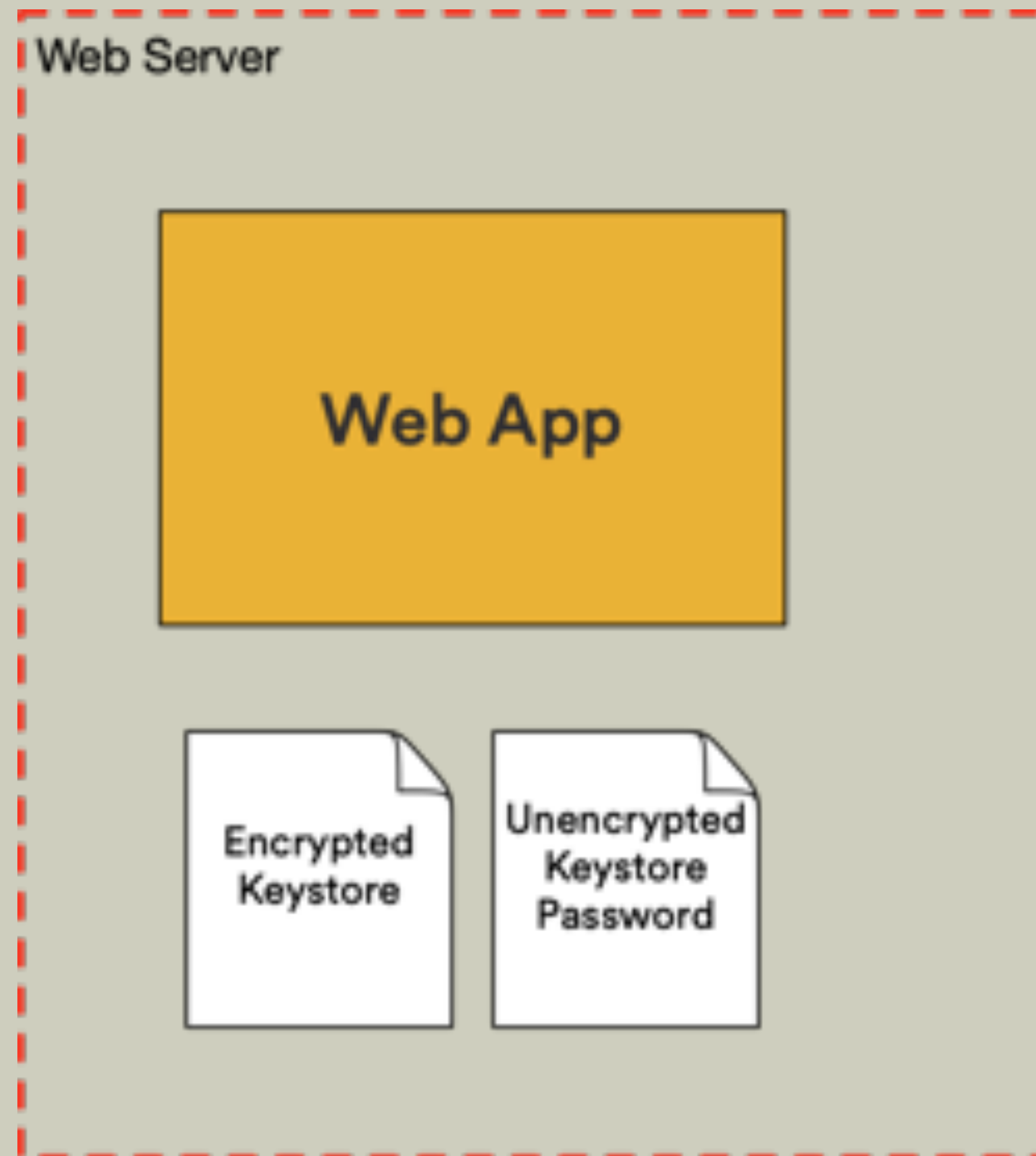
illuminated

Encrypted Keysets

and Key Management Systems

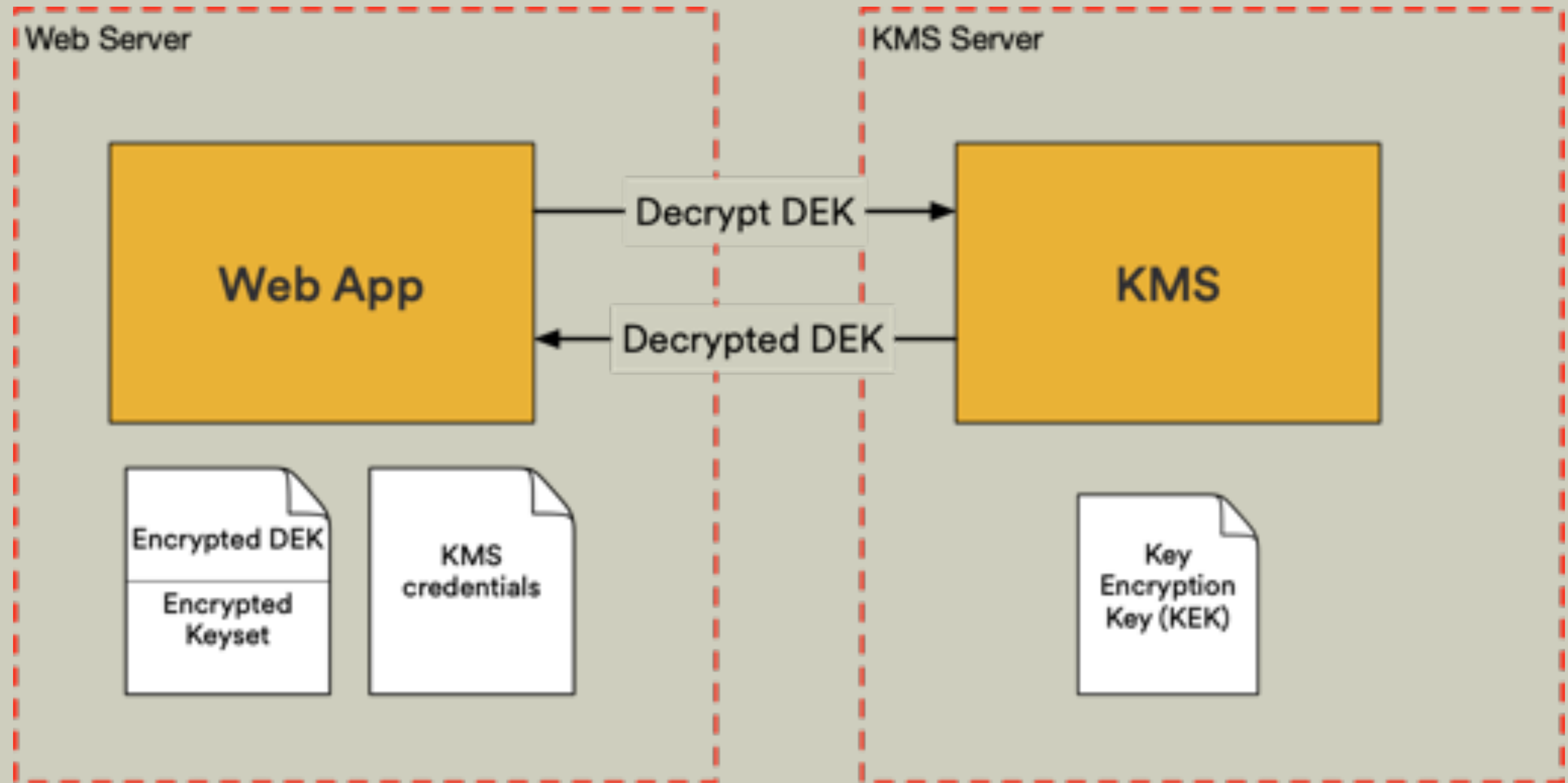
illuminated

Password next to keystore



illuminated

Key Management Systems



illuminated

Supported KMSes

- Google Cloud KMS
- AWS KMS
- Hashicorp Vault (Go only)
- Android Keystore (Java)
- iOS Keychain (Obj-C)

© Kelly Michals/Flickr

Generating an encrypted keyset

```
tinkey create-keyset \
    --key-template AES256_GCM \
    --out-format json \
    --out encrypted-keys.json \
    --master-key-uri gcp-kms://... \
    [--credential kms-credentials.json]
```

Why not use the KMS directly?

© Toni Verdú Carbó

© a4gpa

Auditability

```
InsecureSecretKeyAccess
```

- Required anywhere raw key material is manipulated
- These are key areas for security review
- IDE: Find All Usages

illuminated

Streaming encryption

for larger data

illuminated

Java's CipherInputStream

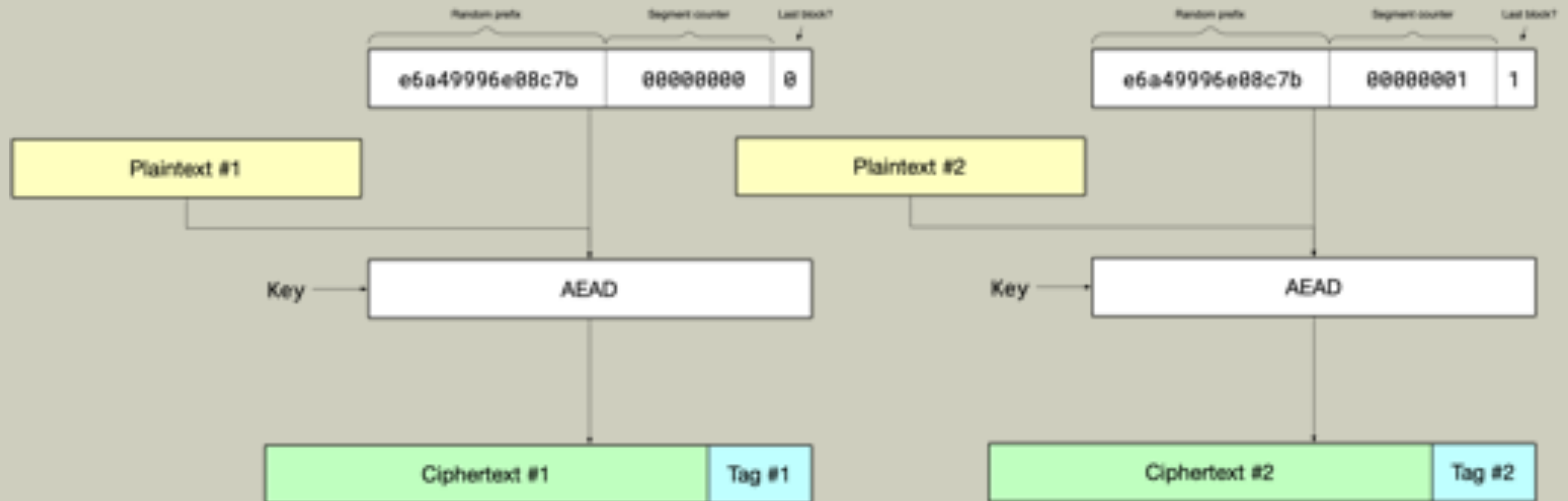
```
try (var in = new CipherInputStream(
        new FileInputStream(path), cipher))
{
    while (in.read(buffer) > -1) {
        process(buffer);
    }
} catch (IOException e) {
    logger.warning(e);
}
```

Releasing unverified plaintext

```
try (var in = new CipherInputStream(...)) {
    while (in.read(buffer) > -1) {
        process(buffer);
    }
}
```

When does this data get authenticated?

Streaming AEAD



illuminated

Streaming AEAD

```
var aead = keyset.getPrimitive(
    StreamingAead.class);
try (var in =
      aead.newDecryptingStream(...)) {

    while (in.read(buffer) > -1) {
        process(buffer);
    }

} catch (IOException e) { ... }
```

illuminated

Streaming AEAD algorithms

Algorithm[1]	Pros	Cons
AES-CTR+HMAC	Simple, robust	Slower
AES-GCM+HKDF	Fast	Fragile-ish

1. Both available in 4KiB and 1MiB variants and 128- or 256-bit keys

Message Authentication

and Signatures

illuminated

Compute a MAC

```
var mac = keyset.getPrimitive(Mac.class);
var tag = mac.computeMac(data);
```

Verifying

```
var mac = keyset.getPrimitive(Mac.class);
mac.verifyMac(tag, data);
```

MAC algorithms

Algorithm	Pros	Cons
HMAC-SHA2	Robust, very secure	Relatively slow
AES-CMAC	Relatively fast	Less common

illuminated

Signing

```
PublicKeySign signer =
    keyset.getPrimitive(PublicKeySign.class);
var signature = signer.sign(data);
```

Verification

```
PublicKeyVerify verifier =
    keyset.getPrimitive(
        PublicKeyVerify.class);
verifier.verify(signature, data);
```

Signature Algorithms

Algorithm	Pros	Cons
ECDSA	Small signatures, widely used	Slow(ish), fragile
Ed25519	Small signatures, fast	Less widely implemented
RSA-3072	Fastest verification	Slow signing speed, large sigs

illuminated

JWTs
illuminated

Creating a JWT

```
var now = Instant.now();
RawJwt rawJwt = RawJwt.newBuilder()
        .setIssuer("https://issuer.example/")
        .setIssuedAt(now)
        .setExpiration(now.plus(10, MINUTES))
        .setTypeHeader("x-jwt-example")
        .build();

JwtMac jwtSigner =
    keyset.getPrimitive(JwtMac.class);
String jwt =
    jwtSigner.computeMacAndEncode(rawJwt);
```

Verifying a JWT

```
var validator = JwtValidator.newBuilder()
    .expectIssuer(
        "https://issuer.example/")
    .expectTypeHeader("x-jwt-example")
    .setClockSkew(Duration.ofMinutes(2))
    .build();
var jwtVerifier =
    keyset.getPrimitive(JwtMac.class);

VerifiedJwt verified =
    jwtVerifier.verifyMacAndDecode(
        jwt, validator);
```

illuminated

Public key signed JWTs

Signing

```
var jwtSigner =
    keyset.getPrimitive(JwtPublicKeySign.class);
String jwt =
     jwtSigner.signAndEncode(rawJwt);
```

Verification

```
var jwtVerifier =
        keyset
            .getPublicKeysetHandle()
            .getPrimitive(JwtPublicKeyVerify.class);
VerifiedJwt verified =
        jwtVerifier.verifyAndDecode(jwt, validator);
```

Supported algorithms

- HMAC: `HS256`, `HS384`, `HS512`
- ECDSA: `ES256`, `ES384`, `ES512`
- RSA: `RS256`, `RS384`, `RS512`, `PS256`, `PS384`, `PS512`

Not supported:

- Encrypted JWTs
- `EdDSA` signatures

illuminated

Publishing public keys as JSON Web Keys

```
String jwks =
    JwkSetConverter.fromPublicKeysetHandle(
        keyset.getPublicKeysetHandle());
System.out.println(jwks);
```

```
{"keys":[{
  "kty":"EC",
  "crv":"P-256",
  "x":"DYC8gjX5p9BoVak2U8f1ovKOewktzODi4rsgv3-I0vc",
  "y":"F6tzT7oKH6qXT8qdJhIb9Ou49KhdZEM6_7973vuuY7s",
  "use":"sig",
  "alg":"ES256",
  "key_ops":["verify"],
  "kid":"qqhSFg"
}]}
```

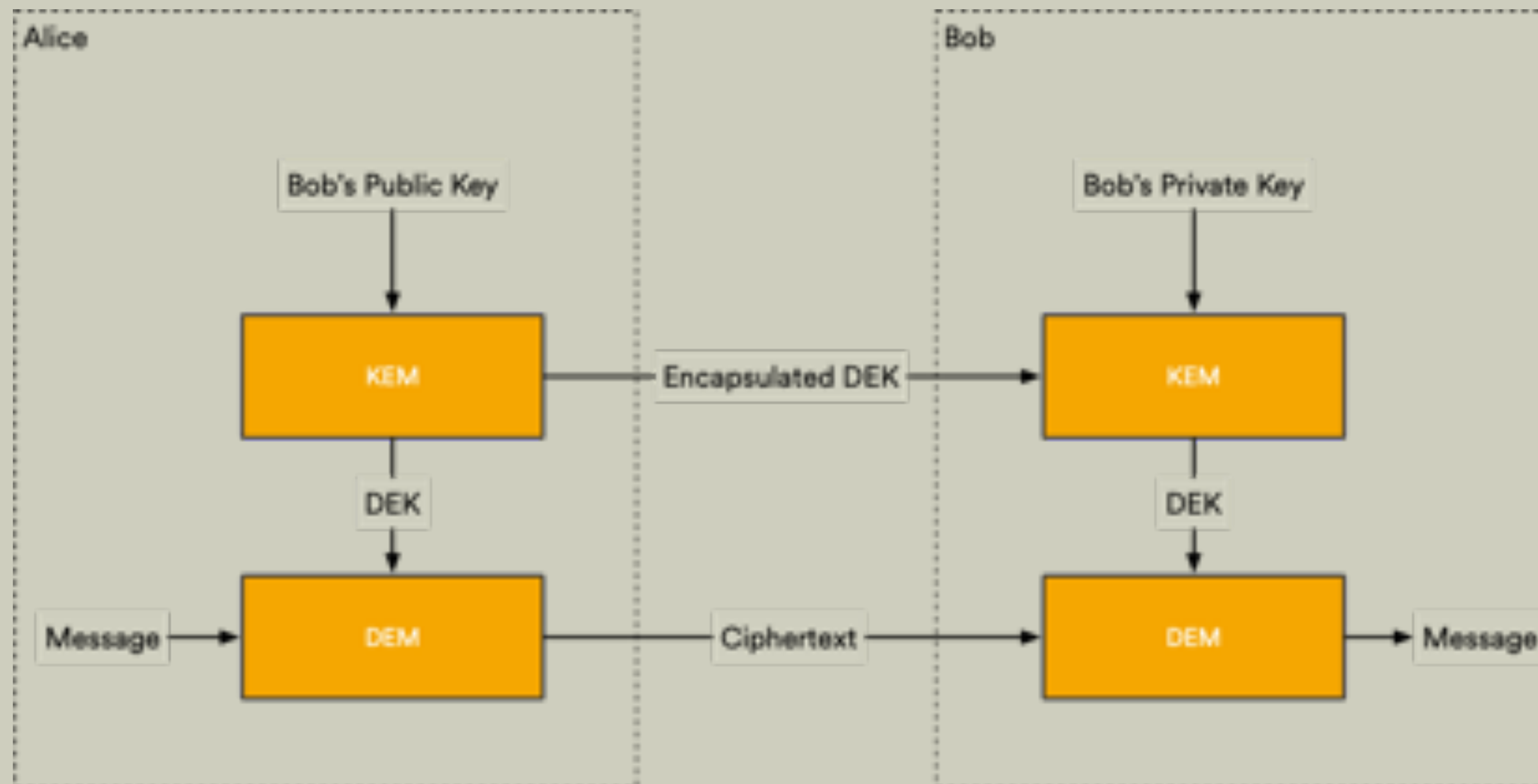
illuminated

Public key encryption

illuminated

Hybrid encryption

and the KEM-DEM paradigm



illuminated

Encrypting a message

```
var encryptor =
  keyset.getPrimitive(HybridEncrypt.class);
var ciphertext =
  encryptor.encrypt(plaintext, ad);
```

Decrypting

```
var decryptor =
  keyset.getPrimitive(HybridDecrypt.class);
var plaintext =
  decryptor.decrypt(ciphertext, ad);
```

Public key encryption algorithms

Algorithm	Pros	Cons
ECIES	Simple	Non-standard
HPKE	More complex	Standard

Limitations: single-recipient, no authenticated KEMs

illuminated

Summary

Easy to use

Hard to *misuse*

Cryptographic agility

Easy to audit

Odds and ends

illuminated

Deterministic AEAD

AES256-SIV

```
var daead = keyset.getPrimitive(
    DeterministicAead.class);
var ciphertext = daead
    .encryptDeterministically(data, ad);
var plaintext = daead
    .decryptDeterministically(ciphertext,
ad);
```

illuminated