OpenSSL CONFERENCE

Jipher FIPS cryptography provider

How we wrapped OpenSSL and the FIPS module using Java's FFM API

Jon Spillett

Copyright © 2026, Oracle and/or its affiliates

Agenda

1. Introduction to Jipher
2. Java's FFM API
3. Architecture and component boundaries
4. Provider and security design
5. Calling OpenSSL from Java
6. Managing native memory
7. Performance design choices
8. Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Agenda

1 **Introduction to Jipher**
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Introduction to Jipher

Why Jipher?

FIPS 140-3 cryptography	Provide FIPS 140-3 compliant cryptography using a validated OpenSSL FIPS module.
JDK compatibility	Designed as a drop-in SunJCE replacement for supported algorithms, with FIPS restrictions and documented provider-specific behaviour.
Explicit restrictions	Apply configured algorithm, key-strength and usage policies.
A Java-maintained binding	Call OpenSSL through Java FFM bindings, and reuse native cipher contexts to reduce allocation overhead.

Copyright © 2026, Oracle and/or its affiliates

Introduction to Jipher

Symmetric cryptography and supporting services

Service	Algorithms, key sizes and modes
AES encryption	128 / 192 / 256-bit keys ECB, CBC, CTR, CFB, OFB and GCM
AES key wrapping	KW and KWP with 128 / 192 / 256-bit keys
Hashing	SHA-1, SHA-2 and SHA-3 SHA-2 / SHA-3 digest sizes: 224 / 256 / 384 / 512 bits
Message authentication	HMAC-SHA1 / SHA224 / SHA256 / SHA384 / SHA512
Key derivation	PBKDF2 with HMAC-SHA1 / SHA224 / SHA256 / SHA384 / SHA512 HKDF-SHA256 / SHA384 / SHA512
Password-based encryption	PBES2 with AES-128 or AES-256 in CBC mode
Random generation	SecureRandom backed by OpenSSL's DRBG

AES padding: NoPadding in all listed modes. ECB and CBC also support PKCS5Padding.

Copyright © 2026, Oracle and/or its affiliates

Introduction to Jipher

Public-key algorithms and key sizes

Family	Operations	Key sizes / parameter sets
RSA	OAEP encryption and decryption PKCS#1 v1.5 and PSS signatures	Generation ≥2048 bits (FIPS) Keygen default: 3072 bits
Elliptic curves	ECDSA signatures ECDH key agreement	P-224 / P-256 / P-384 / P-521 Keygen default: P-256
Diffie–Hellman	Key agreement	Built-in groups: 2048 / 3072 / 4096 bits Keygen default: 3072 bits
ML-KEM*	Key encapsulation	512 / 768 / 1024 parameter sets Keygen default: ML-KEM-768
ML-DSA*	Digital signatures	44 / 65 / 87 parameter sets Keygen default: ML-DSA-65

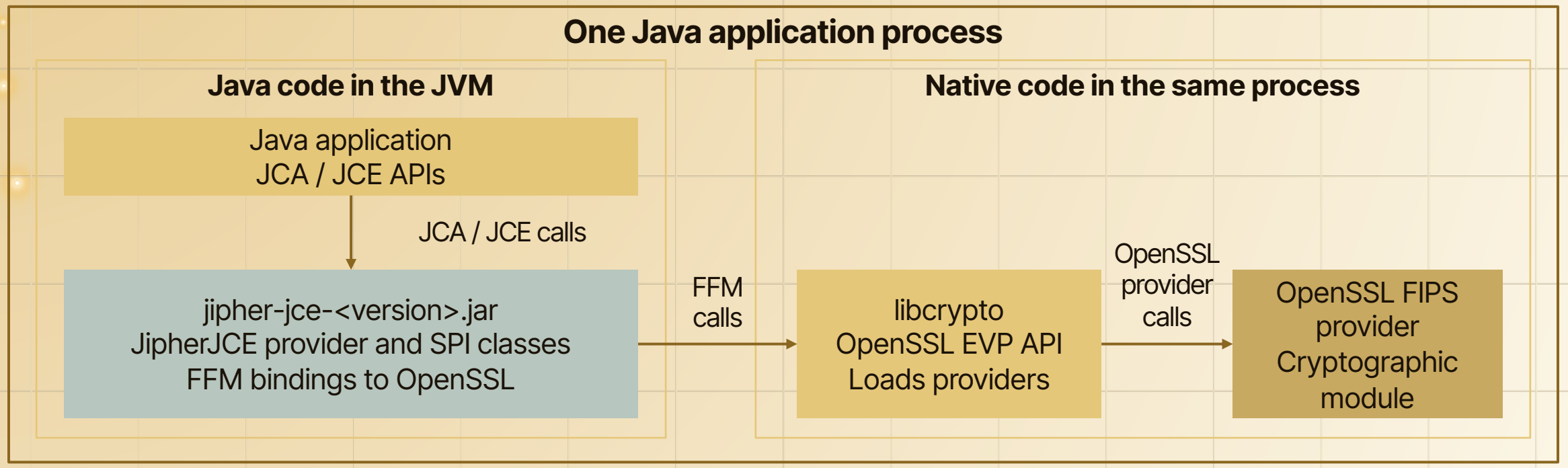
* PQC services require a supporting OpenSSL FIPS module.

Conditional legacy use: DSA and SHA-1 signature verification, plus DESede decryption.

Copyright © 2026, Oracle and/or its affiliates

Introduction to Jipher

Jipher components and runtime placement



Java deployment
JAR on the class path or module path

OpenSSL installation on the host
libcrypto and FIPS module loaded from disk

Copyright © 2026, Oracle and/or its affiliates

Introduction to Jipher

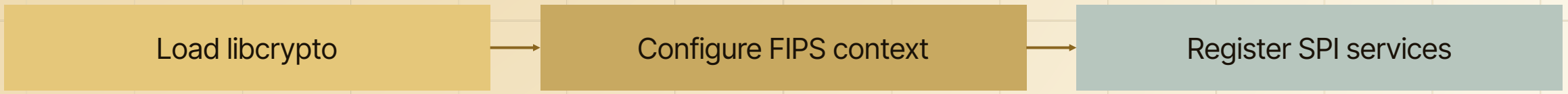
Configuring and initializing Jipher

JDK 25+ and OpenSSL 3.x with an installed FIPS provider. Enable native access for Jipher.

Provider registration

Add JipherJCE to java.security or register it in application code. Provider order controls preference.

Property	Purpose	Default / location
jipher.openssl.dir	Root directory for libcrypto, the FIPS module and configuration	Linux/macOS: /opt/jipher/openssl Windows: C:\Program Files\jipher\openssl
jipher.openssl.useOsInstance	OS installation (Oracle Linux 9.4+) Overrides jipher.openssl.dir	Off by default OS library: /usr/lib64/libcrypto.so.3



Copyright © 2026, Oracle and/or its affiliates

Agenda

1 Introduction to Jipher

2 Java's FFM API

3 Architecture and component boundaries

4 Provider and security design

5 Calling OpenSSL from Java

6 Managing native memory

7 Performance design choices

8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Java's FFM API

Foreign Function and Memory (FFM) in Jipher

Calling OpenSSL from Java

SymbolLookup
Resolve a libcrypto symbol

→

Linker + FunctionDescriptor
Bind the native signature

→

MethodHandle
Invoke the bound function

Upcalls: Linker exposes Java callbacks to OpenSSL, for example, error reporting.

Memory: allocation, lifetime and data representation

Arena
Lifetime and thread-access policy

MemorySegment
View of heap or native memory

MemoryLayout / ValueLayout
Describe data and native types
VarHandle reads / writes fields

FFM checks Java memory access for bounds, lifetime and permitted threads.
Jipher supplies correct C signatures, buffer sizes and ownership rules.
FFM does not check pointer accesses performed inside OpenSSL.

Copyright © 2026, Oracle and/or its affiliates

Java's FFM API

Why Jipher uses FFM

Jipher needs a bridge from Java's JCA/JCE APIs to OpenSSL's C API.
FFM lets Jipher implement that bridge in Java.

No separate JNI glue library

FfmOpenSsl binds directly to libcrypto functions. The binding code ships in Jipher's JAR, without a Jipher C/C++ library to compile and distribute.

JDK handles the platform ABI

FFM implements the platform's application binary interface (ABI): calling conventions and C type layouts. Jipher reuses its Java bindings across supported operating systems and CPUs.

Control over calls and buffers

Cryptographic operations repeatedly cross this boundary. EvpCipherCtx uses FFM to control buffer access and call overhead, while keeping memory handling in Java.

Copyright © 2026, Oracle and/or its affiliates

Agenda

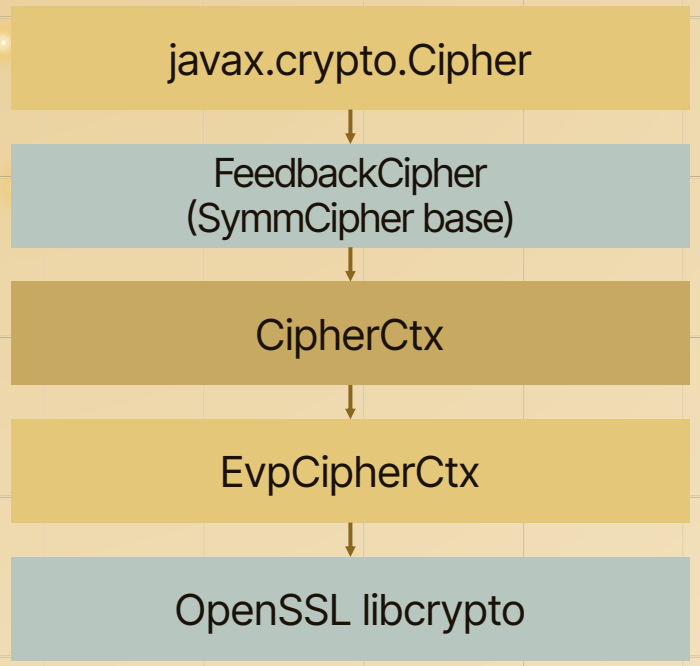
1 Introduction to Jipher
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Architecture and component boundaries

The Java-to-OpenSSL architecture

Example: Cipher.update(...) on an initialized AES/CBC cipher



CipherSpi is the service provider interface (SPI) base class.
Cipher dispatches the call to the selected implementation.

Jipher SPI: checks initialization and output size.
The inherited updateInternal method handles buffer overlap.

Java adapter: forwards update through an EVP_CIPHER_CTX interface reference, without exposing the native pointer to the SPI.

FFM implementation: creates buffer views, checks native capacity and invokes the stored downcall MethodHandle.

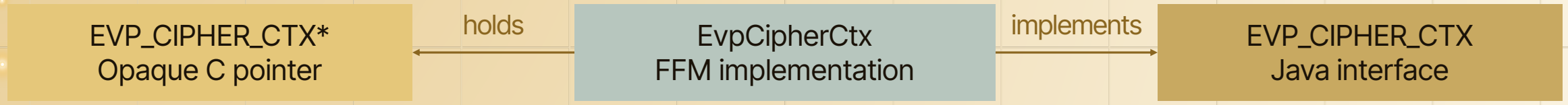
Native execution: EVP_CipherUpdate runs the cipher operation using the OpenSSL provider selected for the context.

Binding setup: OpenSslLoader opens libcrypto. FfmOpenSsl resolves symbols and builds reusable call handles. The JDK linker performs the native transition when a handle is invoked.

Copyright © 2026, Oracle and/or its affiliates

Architecture and component boundaries

Typed Java wrappers for opaque OpenSSL objects



The Java method names the operation

Java method	OpenSSL function
update(...)	EVP_CipherUpdate(ctx, ...)
doFinal(...)	EVP_CipherFinal_ex(ctx, ...)
reset()	EVP_CIPHER_CTX_reset(ctx)

```
// EvpCipherCtx supplies its stored pointer as the first C argument.
EVP_CIPHER_UPDATE_HEAP_FUNC.invokeExact(
    this.evpCipherCtx, out, this.outLenSeg, in, (int) inLen);
```

Copyright © 2026, Oracle and/or its affiliates

Architecture and component boundaries

Algorithm metadata and operation state

Share algorithm metadata. Give each active operation its own mutable context.

EVP_CIPHER / EvpCipher
Cached algorithm metadata

CipherCtx collects algorithm objects once.
EvpCipher caches sizes, flags and mode for reuse.

```
// EvpCipher.blockSize(): cached value
return this.blockSize;
```

EVP_CIPHER_CTX / EvpCipherCtx
Mutable operation state

A context holds the operation's state.
Concurrent operations need separate contexts.

```
// EvpCipherCtx.blockSize(): initialized path
return (int)
    EVP_CIPHER_CTX_GET_BLOCK_SIZE_FUNC
        .invokeExact(this.evpCipherCtx);
```

The same Java method name can mean a cached read or a native call.

Copyright © 2026, Oracle and/or its affiliates

Agenda

1 Introduction to Jipher
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Provider and security design

Two layers govern cryptographic operations

Java checks each requested operation. LibCtx selects FIPS implementations in a private OpenSSL context.

Java SPI + Fips
Operation, algorithm,
key strength and parameters

→

Private OSSL_LIB_CTX
FIPS provider + fips=yes

LibCtx.isFips() verifies the configured native context

Startup request	Required result
Fetch SHA-256	Implementation comes from fips
Fetch MD5 with and without the FIPS property filter	Unavailable in this context

Startup failure prevents Jipher provider initialization.

Copyright © 2026, Oracle and/or its affiliates

Provider and security design

How Jipher builds its service catalogue

The loaded FIPS module and Jipher's policy determine which services JCA can select.

Loaded FIPS module
Capabilities reads its
name and version

Jipher enforcement policy
Startup setting controls
permitted algorithm families

→ JipherJCE.addServices()
Registers names, aliases
and SPI factories

Operation-specific checks still run
when a service is used.

Example	Registration condition
ML-DSA	Loaded FIPS module passes the capability check
DSA / DESede	Excluded when FIPS_STRICT is selected

Copyright © 2026, Oracle and/or its affiliates

Provider and security design

Java policy depends on the operation

Set jipher.fips.enforcement before Jipher initializes.

Mode	Java policy
FIPS — default	Acceptable + legacy-use rules
FIPS_STRICT	Acceptable-use rules only
NONE	Skip Java policy checks; native checks remain

Example: a 1024-bit RSA key — Java policy result

Operation	FIPS	FIPS_STRICT
Verify an existing signature	Passes Java check	Rejected
Create a new signature	Rejected	Rejected

DigestSignature checks initSign / initVerify; policy rejection → InvalidKeyException.

Copyright © 2026, Oracle and/or its affiliates

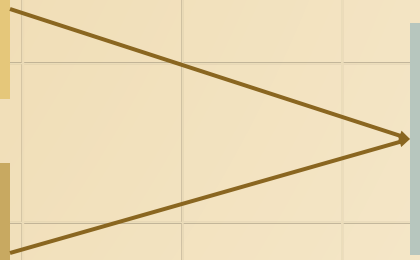
Provider and security design

Where Jipher's randomness comes from

Jipher uses OpenSSL for random generation through both of these routes.

Jipher cryptographic operations
Example: AES key generation

Jipher SecureRandom
nextBytes and generateSeed



OpenSSL random subsystem
Jipher's private library context
Native FIPS random generation

Application-supplied SecureRandom: accepted by the API but ignored as a source of bytes.

Design choice: use the native FIPS random generator
and avoid callbacks to an application-selected Java RNG.

Copyright © 2026, Oracle and/or its affiliates

Agenda

1 Introduction to Jipher
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Calling OpenSSL from Java

1 Describe and locate the function

Jipher needs the function's native signature and its address in libcrypto.

Start with the C function signature

```
int EVP_CipherUpdate(
    EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl,
    const unsigned char *in, int inl);
```

Describe its argument and return types

"(MMMMI)I" is Jipher's shorthand: four pointers, an input length, and an int status result.
FfmOpenSsl converts it to a FunctionDescriptor. M means pointer. I means C int.

Find the function's address

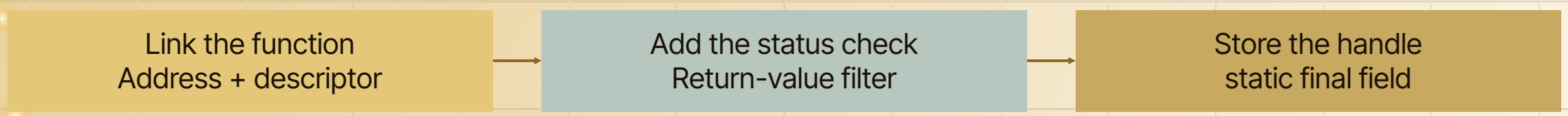
FfmOpenSsl uses LIBCRYPTO.find("EVP_CipherUpdate") to look up the exported symbol.
The address and descriptor are the inputs to the native linker.

Copyright © 2026, Oracle and/or its affiliates

Calling OpenSSL from Java

2 Build and cache the call handle

FfmOpenSsl creates the checked handle. EvpCipherCtx stores it for later updates.




```
// In EvpCipherCtx: bind the variant that accepts Java arrays.
static final MethodHandle EVP_CIPHER_UPDATE_HEAP_FUNC;

static {
   EVP_CIPHER_UPDATE_HEAP_FUNC = downcallHandleCheckZeroNeg(
      "EVP_CipherUpdate", "(MMMMI)I", HEAP_ACCESS);
}
```

Binding runs once during class initialization. The static final field also gives the JIT a constant handle target, allowing it to optimize the Java call adapters.

Copyright © 2026, Oracle and/or its affiliates

Calling OpenSSL from Java

3 Call OpenSSL with the current buffers

For an initialized cipher using byte[] buffers, EvpCipherCtx.update(...) does three things.

Prepare the buffers

Wrap the input and output arrays as MemorySegment views.
Check that the output buffer has enough capacity.

Invoke the handle

```
// One chunk from the heap-buffer path.
EVP_CIPHER_UPDATE_HEAP_FUNC.invokeExact(
    this.evpCipherCtx, out, this.outLenSeg, in, (int) inLen);
```

Collect the output

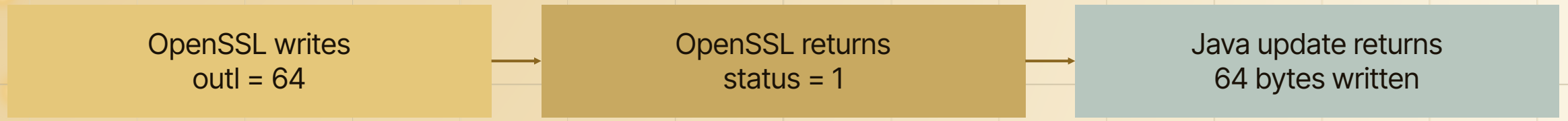
After a successful call, read the byte count from outLenSeg.
Advance the buffer slices, repeat for any remaining input,
and return the total number of bytes written.

Copyright © 2026, Oracle and/or its affiliates

Calling OpenSSL from Java

4 Separate status, output length and Java failure

Example: a fresh AES/CBC/NoPadding update receives 64 input bytes.



The checked MethodHandle consumes the C status and returns void

```
// FfmOpenSsl status filter, used for positive-success functions.
if (returnValue <= 0) {
    throw newOpenSslException(funcName);
}
```

On failure, EvpCipherCtx maps the exception instead of returning an output count.
The status code and the number of bytes written are separate results.

Copyright © 2026, Oracle and/or its affiliates

Agenda

1 Introduction to Jipher
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Managing native memory

Four responsibilities at the native boundary

A native pointer raises four separate questions.

Bounds

How many bytes may this call read or write?

Ownership

Who must release the native reference?

Lifetime

Until when is this pointer valid?

Zeroization

Which copies of a secret are cleared, and when?

An arena controls lifetime. Bounds, ownership and secret clearing still need explicit design.

Copyright © 2026, Oracle and/or its affiliates

Managing native memory

Adopting an opaque OpenSSL handle

EVP_CIPHER_CTX needs a scoped handle, not a Java layout of its internals.

EVP_CIPHER_CTX_new()
OpenSSL allocates the object

→

reinterpret(arena, cleanup)
Java scopes the pointer view

→

Arena cleanup runs
EVP_CIPHER_CTX_free()

```
// Keep the address; register the matching OpenSSL destructor.
this.evpCipherCtx = evpCipherCtx.reinterpret(arena, EvpCipherCtx::free);
```

The handle and reusable scratch slices share an arena.

No object copy occurs. Closing an explicit arena invalidates those segments.
Caller input/output buffers retain their own ownership and lifetime.

Copyright © 2026, Oracle and/or its affiliates

Managing native memory

Owned references and borrowed handles

The OpenSSL API determines whether a Java wrapper must release a reference.

Owned: fetchCipher(...)	Borrowed: forEachCipher(...)
Register EVP_CIPHER_free. Arena cleanup releases this reference.	The enumeration wrapper has no free action. It does not own the native reference.

To keep a cipher beyond enumeration, retain a reference while the handle is valid:

Borrowed cipher inside the callback → EvpCipher.upRef() EVP_CIPHER_up_ref → New owned wrapper matching free action

reinterpret() alone does not increment the native reference count.

Copyright © 2026, Oracle and/or its affiliates

Managing native memory

Allocation policy and secret clearing

OpenSslAllocators combines native allocation, a sized Java view and matching cleanup.

Allocation policy	Initially zeroed	Cleared before free
malloc(size, arena)	No	No
mallocClearFree(size, arena)	No	Yes
zallocClearFree(size, arena)	Yes	Yes

```
// Record the size and cleanup in the scoped view.
segment.reinterpret(byteSize, arena, OpenSslAllocators::free);
```

FFM checks Java memory access. Wrappers must validate capacity before native writes.

Copyright © 2026, Oracle and/or its affiliates

Managing native memory

Validate key and IV lengths before native access

EVP_CipherInit_ex2 receives pointers, not the sizes of the Java arrays.

Supplied IV: 15 bytes

AES-CBC requires 16 bytes

Reject the short IV before copying it or calling OpenSSL.

FfmOpenSsl.allocateFromNullable(...)

```
if (bytes != null && bytes.length != requiredLen) {
    throw new IllegalArgumentException(...);
}
```

Copyright © 2026, Oracle and/or its affiliates

Managing native memory

Output capacity must be checked before the call

EvpCipherCtx.update(): one new AES-CBC encryption byte can complete a block.

Destination: 1 byte
Throw ShortBufferException

Destination: at least 16 bytes
Enough capacity for this update

EVP_CipherUpdate(ctx, out, &outl, in, inl);

outl reports bytes written. This API has no output-capacity argument.

Copyright © 2026, Oracle and/or its affiliates

Agenda

1 Introduction to Jipher
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

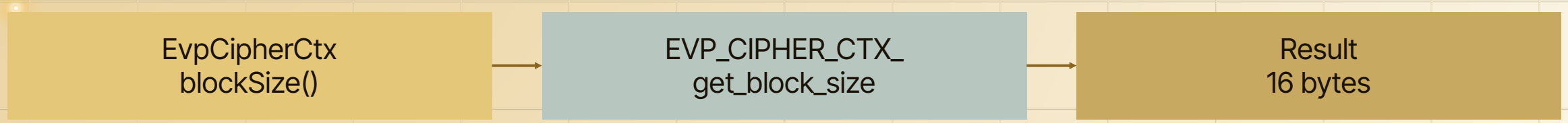
Copyright © 2026, Oracle and/or its affiliates

Performance design choices

Case study: critical downcalls for tiny operations

A critical downcall tells the JVM that native work is extremely short and never calls back into Java. The JVM can reduce call overhead.

Example: reading the block size of an initialized AES/CBC cipher




```
// CRITICAL = critical(false): critical call with heap access disabled.
downcallHandle("EVP_CIPHER_CTX_get_block_size", INT_FUNCDESC, CRITICAL);
```

The call boundary can be a significant cost when native code only reads a value.
Long critical calls can delay JVM safepoints, including garbage collection.

Copyright © 2026, Oracle and/or its affiliates

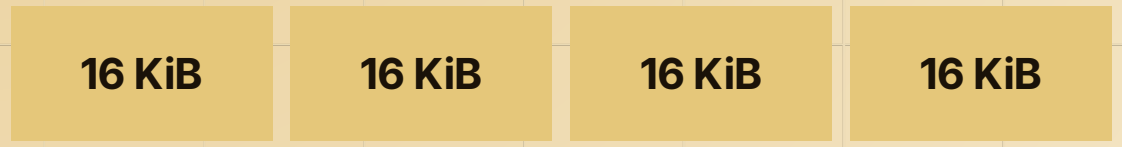
Performance design choices

Case study: heap access with bounded native calls

Example: EvpCipherCtx.update(...) processes 64 KiB for AES/CBC.
Buffer storage selects the handle that calls EVP_CipherUpdate.

Either buffer is on the Java heap

HEAP_ACCESS = critical(true)



4 calls, each limited to 16 KiB

Avoid staging copies at the FFM boundary.
Process at most 16 KiB per critical call.

Both buffers are native

NONE = ordinary downcall

1 call for this input

Native memory needs no heap access.
Larger chunks reduce call count.

Copyright © 2026, Oracle and/or its affiliates

Performance design choices

Case study: reusing a cipher context between messages

Example: completing one AES/GCM message, then initializing the next.

Message A finishes
AES/GCM doFinal()

→

cleanup() releases context
Reset, then return to pool

→

Message B initializes
Acquire from the pool

EvpCipherCtx.getEvpCipherCtxAutoArena()

```
// Reuse a released context, or allocate on a pool miss.
return CIPHER_CTX_OBJECT_POOL.pop()
    .orElseGet(() → new EvpCipherCtx(true, Arena.ofAuto()));
```

Storage can be reused after reset.
Each AES/GCM message still initializes its key and IV.

Copyright © 2026, Oracle and/or its affiliates

Performance design choices

Case study: the allocation work a pool hit avoids

Example workload: repeated 1 KiB AES/GCM messages, each with a fresh init.

Work at the next acquisition	Pool miss	Pool hit
EvpCipherCtx + automatic arena	Create	Reuse
Native EVP_CIPHER_CTX	Allocate	Reuse
FFM scratch buffer and views	Allocate	Reuse
Initialize key/IV and encrypt	Still required	Still required

A hit avoids repeated allocation work, which may matter more for short messages.
The tradeoff is retained memory and atomic pool access. Gains depend on the workload.

Copyright © 2026, Oracle and/or its affiliates

Agenda

1 Introduction to Jipher
2 Java's FFM API
3 Architecture and component boundaries
4 Provider and security design
5 Calling OpenSSL from Java
6 Managing native memory
7 Performance design choices
8 Takeaways and questions

Copyright © 2026, Oracle and/or its affiliates

Takeaways

Jipher's key design choices

FIPS 140-3 through JCA	Familiar Java cryptography APIs backed by a validated OpenSSL FIPS module.
Policy and module capabilities	Module capabilities shape available services. Jipher applies operation-specific key and usage policies.
OpenSSL supplies randomness	A supplied Java SecureRandom does not change the random source used by Jipher operations.
FFM and native memory	Bindings stay in Java. Jipher validates buffers and manages native ownership, lifetime and secret clearing.
Targeted performance choices	Cached handles, selective critical calls and pooled contexts can reduce call, copy and allocation costs.

Copyright © 2026, Oracle and/or its affiliates

Browse the code

OpenJDK Project Brisbane

Explore the implementation behind this talk.

Project Brisbane is part of OpenJDK.
It operates under the guidance of Oracle's JDK team.

github.com/openjdk/brisbane/tree/master

Get started with README.md
More detail in /doc/*

Copyright © 2026, Oracle and/or its affiliates

Questions

OpenSSL CONFERENCE

Thank you

github.com/openjdk/brisbane/tree/master

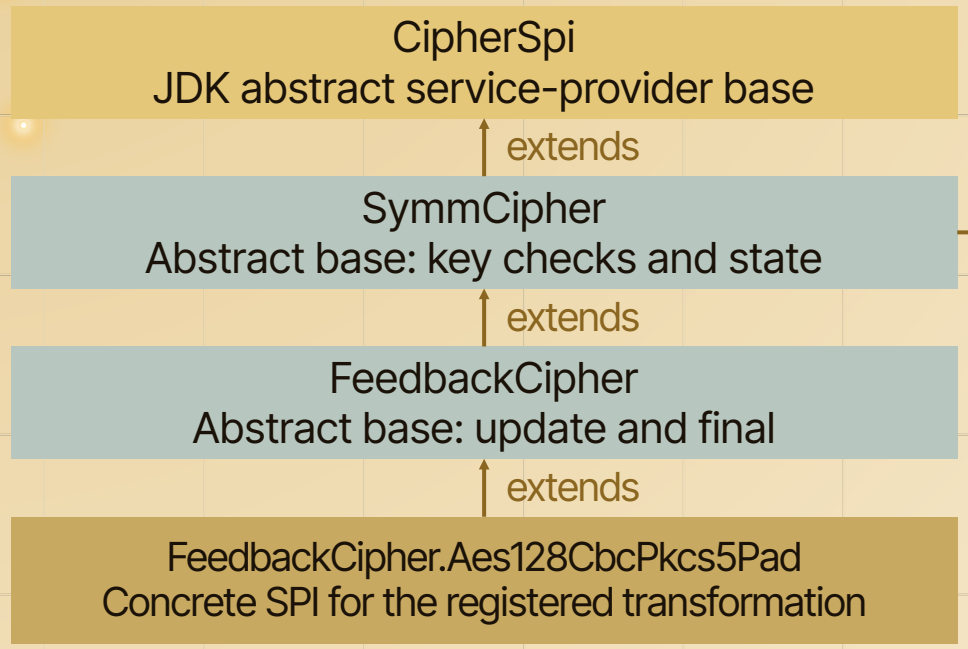
Copyright © 2026, Oracle and/or its affiliates

Extra material

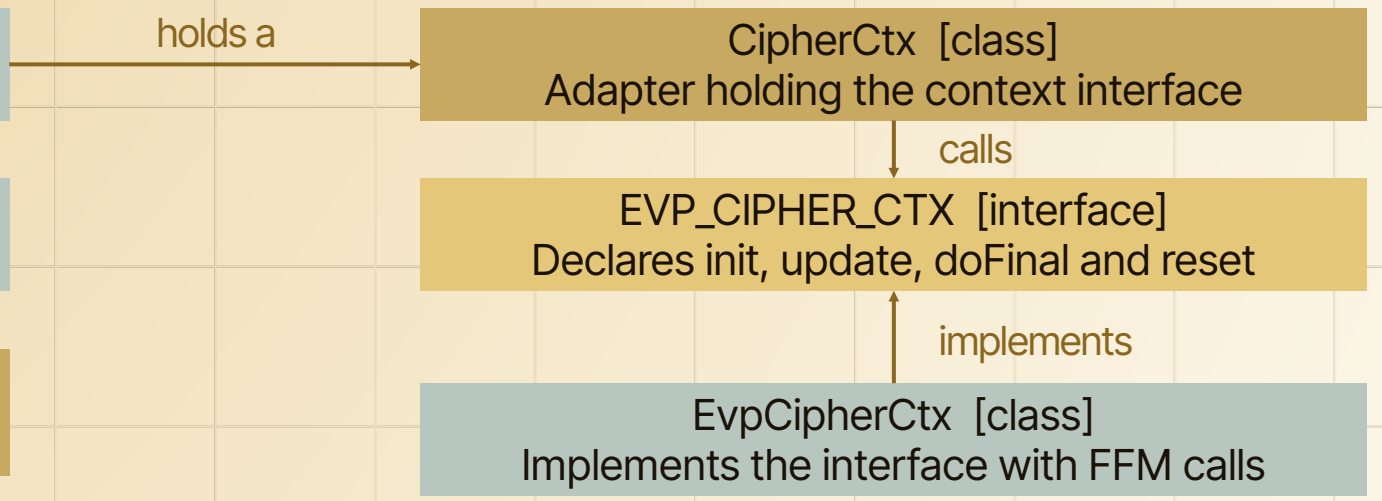
Interfaces and classes in the symmetric cipher path

Example: JipherJCE registers AES_128/CBC/PKCS5Padding with a concrete SPI factory.

SPI class inheritance



Adapter and implementation

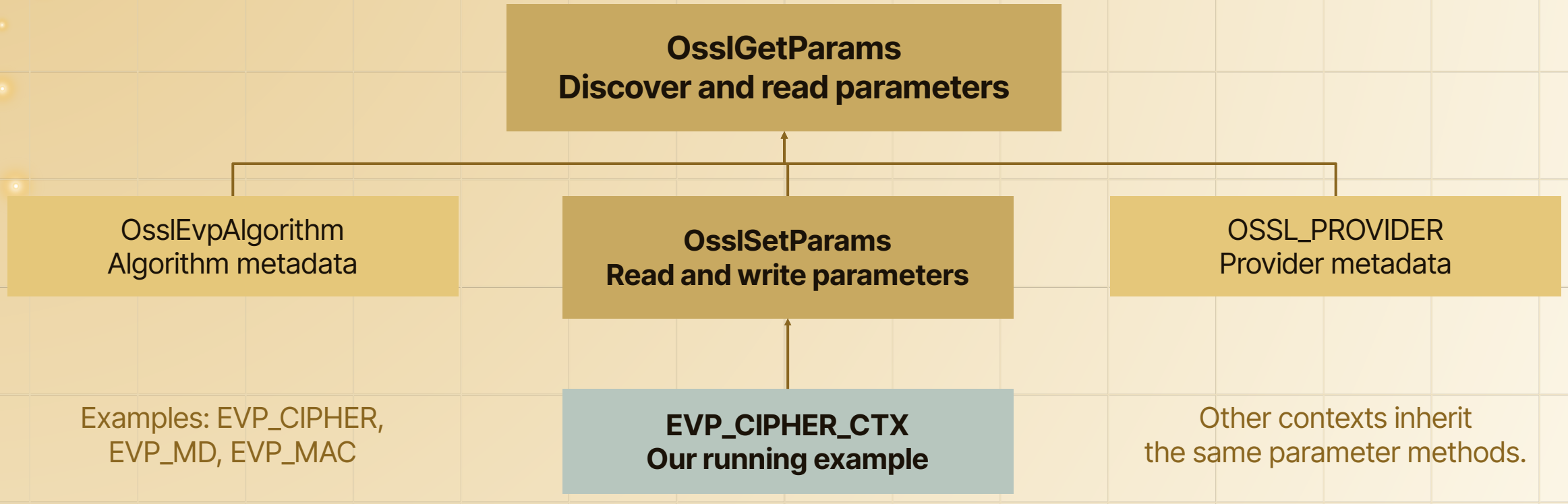


Inheritance shares SPI behavior. The Java interface separates the adapter from its FFM implementation.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Shared parameter interfaces



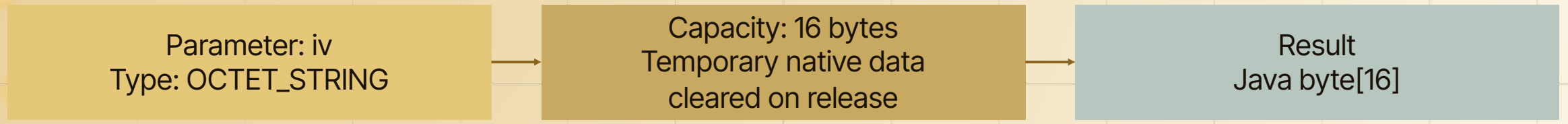
Arrowheads point to parent interfaces.
Java OSSL_PARAM descriptors represent OpenSSL's native OSSL_PARAM structure.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Case study: requesting an IV

Example: CipherCtx.getIv() requests the initialization IV for AES/CBC.



CipherCtx.getIv()

```
int len = this.evpCipherCtx.ivLength();
OSSL_PARAM template = OSSL_PARAM.of(
    EVP_CIPHER.CIPHER_PARAM_IV,
    OSSL_PARAM.Type.OCTET_STRING, len)
    .sensitive();
OSSL_PARAM[] params = this.evpCipherCtx.getParams(template);
return params[0].data;
```

Line-by-line explanation

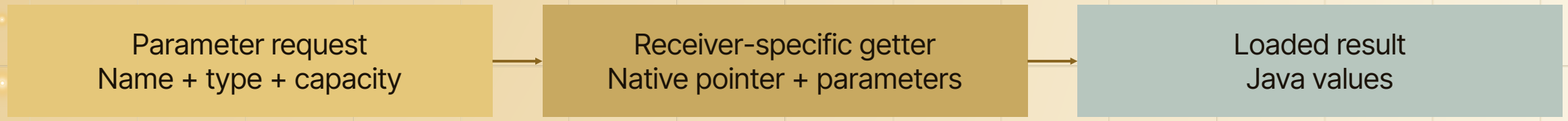
```
// Required IV size in bytes
// Describe the output parameter
// OpenSSL parameter name: "iv"
// Byte-array type and capacity
// Mark native data for clearing
// Ask the context to fill it
// Return the IV as Java bytes
```

OSSL_PARAM describes the request. OpenSSL fills the native output buffer.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Case study: how the shared getter fills the request



OsslGetParams default method

```
default OSSL_PARAM[] getParams(OSSL_PARAM... params) {
    OsslParamBuffer buffer = OpenSsl.getInstance()
        .templateParamBuffer(params);
    getParams(buffer);
    return buffer.asArray();
}
```

Line-by-line explanation

```
// Accept Java parameter requests
// Select the OpenSSL backend
// Prepare the output requests
// Call this object's native getter
// Return populated Java values
// End the shared getter
```

For the IV request: EvpCipherCtx invokes EVP_CIPHER_CTX_get_params(ctx, params).

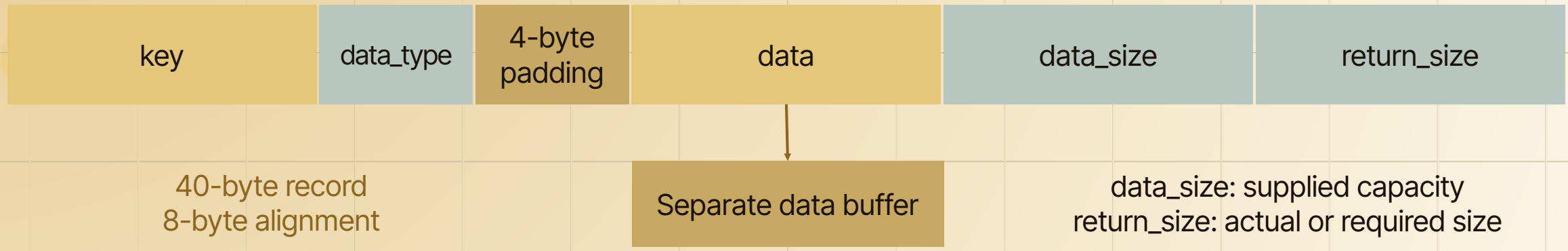
gettableParams discovers available parameters. This getter reads the requested values.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Describing the native OSSL_PARAM layout

OsslParam models OpenSSL's OSSL_PARAM record for the supported 64-bit ABI.




```
// A named field becomes a typed accessor into a MemorySegment.
C_OSSL_PARAM.varHandle(groupElement("data_size"));
```

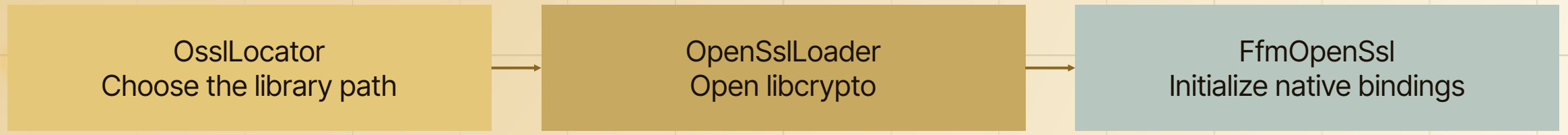
MemoryLayout describes field offsets and alignment; it does not allocate the record or its data.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Loading OpenSSL: selection and library lifetime

OpenSslFactory chooses the installation and publishes the initialized backend.



Installation selection

Configured directory, platform default,
or a supported operating-system installation.

Library lifetime

Arena.global() keeps the library loaded
while its cached handles can be used.

The factory checks the library version. Failed initialization prevents normal provider use.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Loading OpenSSL: initialize the lookup before the bindings

FfmOpenSsl reads the loader's lookup during class initialization.

1 Load libcrypto and retain its lookup

```
// OpenSslLoader.load(...), excerpt
libCrypto = SymbolLookup.libraryLookup(
    libCryptoPath, Arena.global());
```

2 Initialize the binding class

```
// Its static bindings can now resolve symbols.
return new FfmOpenSsl();
```

The synchronized loader guards one initialization per defining class loader.
This ordering gives cached downcalls a live library and a ready symbol lookup.

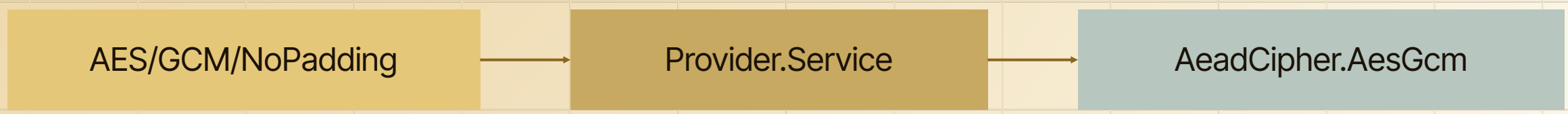
Copyright © 2026, Oracle and/or its affiliates

Extra material

Service registration selects a concrete SPI

A transformation name maps to a factory for its CipherSpi.

```
// In JipherJCE.addServices(): register the exact transformation.
putService("Cipher", "AES/GCM/NoPadding",
    AeadCipher.AesGcm.class.getName(),
    AeadCipher.AesGcm::new);
```



Copyright © 2026, Oracle and/or its affiliates

Extra material

Cipher initialization: policy, parameters and cleanup

Key and policy checks
Type, algorithm and strength

→

Parameter checks
Mode-specific IV / tag rules

→

Initialize CipherCtx
Commit success state

Every exit clears the temporary Java key copy

```
// SymmCipher.initInternal: cleanup runs on success and failure.
finally {
    clearArray(keyData);
    if (!this.initialized) {
        releaseCtx(); // release a context after failed initialization
    }
}
```

A Java policy rejection becomes InvalidKeyException before cipher operation setup.

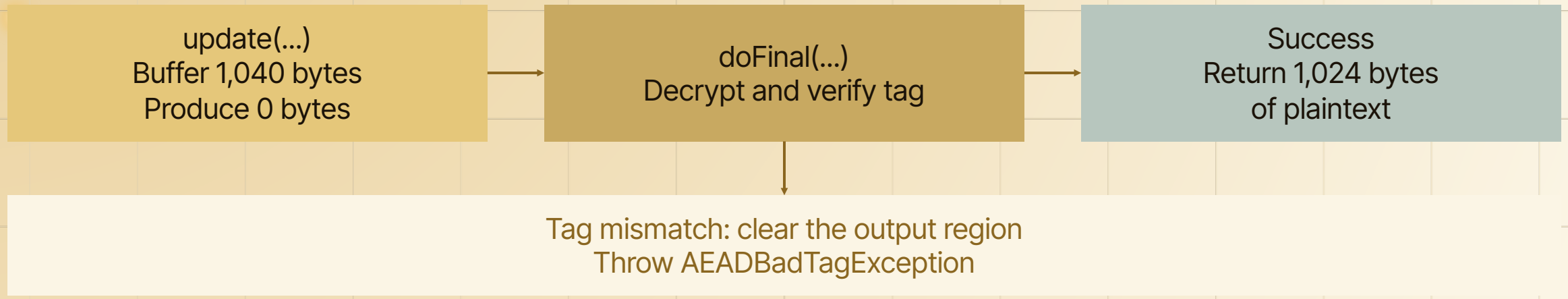
Copyright © 2026, Oracle and/or its affiliates

Extra material

Case study: authenticated AES/GCM decryption

By default, AES/GCM decryption buffers ciphertext supplied to update and returns plaintext only after doFinal verifies the tag.

Example: 1,024-byte ciphertext + 16-byte authentication tag



Default: update returns no unauthenticated plaintext.
Buffer memory grows with update input.

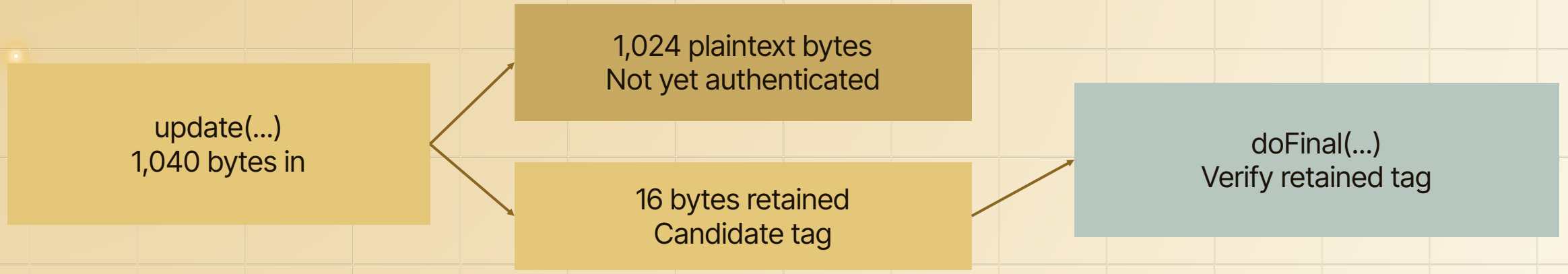
Copyright © 2026, Oracle and/or its affiliates

Extra material

Case study: the AES/GCM streaming-decryption tradeoff

-Djipher.cipher.AEAD.stream=true

Same 1,040-byte input; authentication still finishes in doFinal.



Tag mismatch → ProviderException; discard all recovered plaintext.

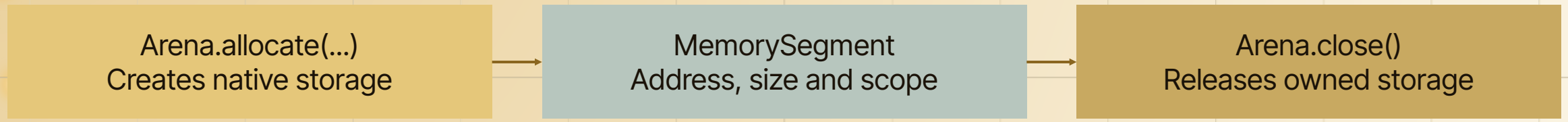
Less provider buffering. Accept the message only after doFinal succeeds.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Temporary native storage with an Arena

EvpPkeyCtx.encapsulate() needs two native size_t output-length slots.




```
try (Arena arena = Arena.ofConfined()) {
    MemorySegment wrappedKeyLenSeg = arena.allocate(C_SIZE_T);
    MemorySegment genKeyLenSeg = arena.allocate(C_SIZE_T);
    // Query sizes and consume the results inside this scope.
}
```

After close, Java access through these segments fails.
Native code must not retain their pointers beyond the scope.

Copyright © 2026, Oracle and/or its affiliates

Extra material

When cleanup happens

Choose cleanup timing separately from ownership and secret clearing.

Mechanism	Trigger	Native cleanup timing
Explicit arena	arena.close()	During close
Automatic arena	Scope becomes unreachable	JVM-chosen time
EvpPkey.free()	Explicit wrapper call	Once; later cleanup is a no-op

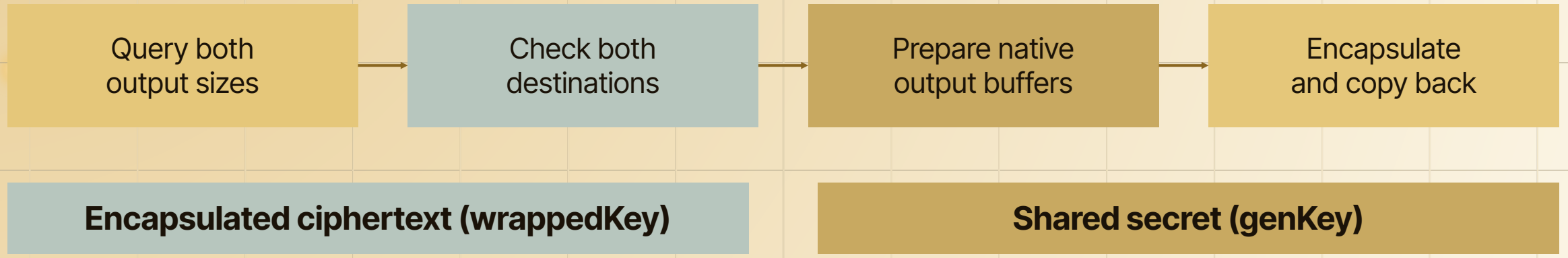
Early key release also marks the wrapper inactive.
Returning a cipher to its pool resets it for reuse; it does not free the native allocation.

Copyright © 2026, Oracle and/or its affiliates

Extra material

KEM output: size, copy and clear

EvpPkeyCtx.encapsulate() produces public ciphertext and a shared secret.



Native destinations can be used directly. Clearing covers the temporary native secret; the caller's output remains available.

Copyright © 2026, Oracle and/or its affiliates

Extra material

Case study: how the shared cipher pool works

Each thread ID selects one of 64 shared stacks to reduce contention.

Acquire a cipher context

A stack hit reuses the context. Other main stacks are not searched.

Return after reset

Try the releasing thread's selected stack, then overflow.
If both are full, the pool does not retain the context.

At most 1,640 idle contexts

64 × 10 main entries + 1,000 overflow entries. Active contexts are additional.

Copyright © 2026, Oracle and/or its affiliates