Post-Quantum

Cryptography Conference

A Framework for Cryptographic Agility

Navaneeth Rameshan
Senior Research Developer at IBM Research

KEYFACTOR CRYPTO4A SSL.com ENTRUST HID

October 28 - 30, 2025 - Kuala Lumpur, Malaysia

PKI Consortium

A Framework For Cryptographic Agility

Navaneeth Rameshan

Senior Research Developer

IBM Research Europe

The Typical Crypto Call

```python
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes

# Generate ECDSA private key
private_key = ec.generate_private_key(ec.SECP256R1())

# Sign some data
data = b"data"
signature = private_key.sign(data, ec.ECDSA(hashes.SHA256()))

# Verify signature
public_key = private_key.public_key()
public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
```

The Typical Crypto Call

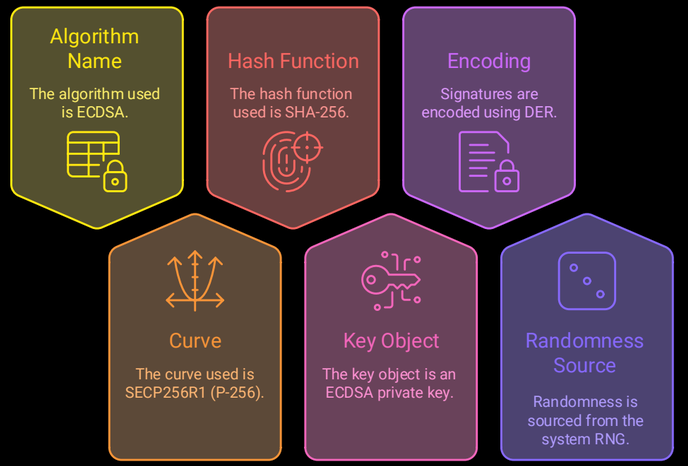
```python
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes

# Generate ECDSA private key
private_key = ec.generate_private_key(ec.SECP256R1())

# Sign some data
data = b"data"
signature = private_key.sign(data, ec.ECDSA(hashes.SHA256()))

# Verify signature
public_key = private_key.public_key()
public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
```

Every call bakes in algorithm specifics

If any one of these inputs change, the application code changes too

The Typical Crypto Call

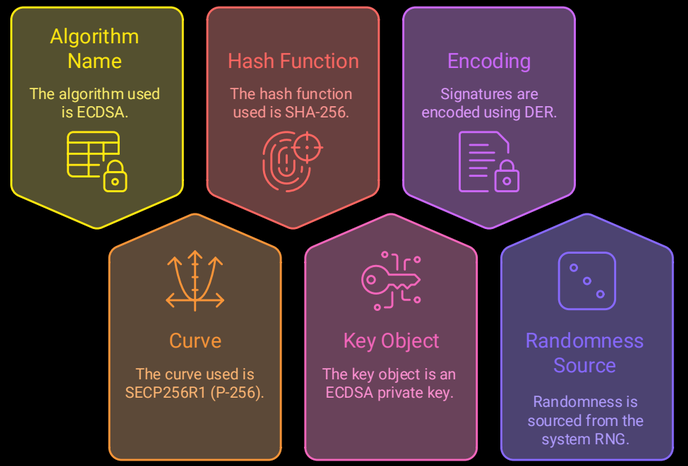
```python
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes

# Generate ECDSA private key
private_key = ec.generate_private_key(ec.SECP256R1())

# Sign some data
data = b"data"
signature = private_key.sign(data, ec.ECDSA(hashes.SHA256()))

# Verify signature
public_key = private_key.public_key()
public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
```



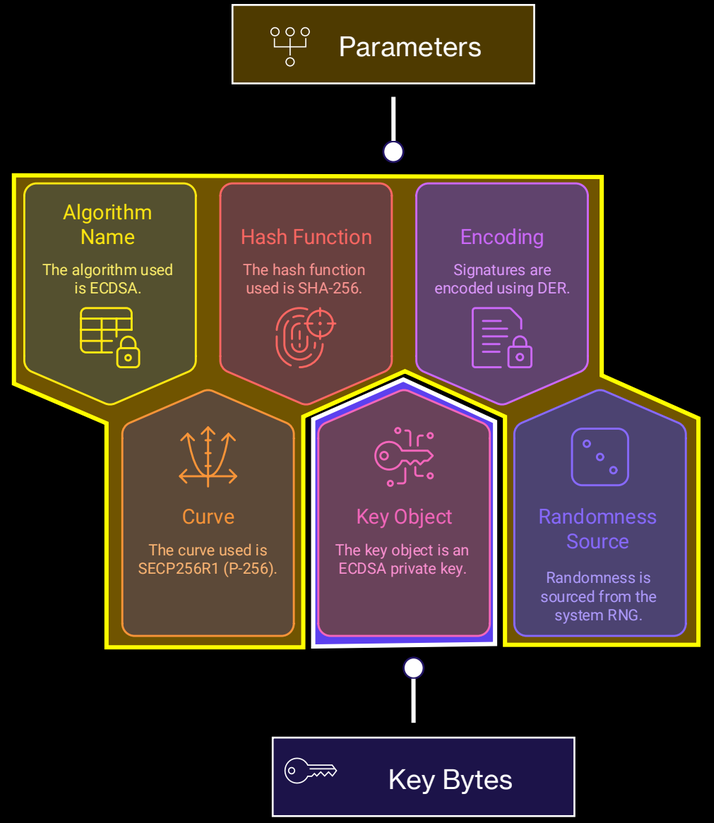
Every call bakes in algorithm specifics

If any one of these inputs change, the application code changes too

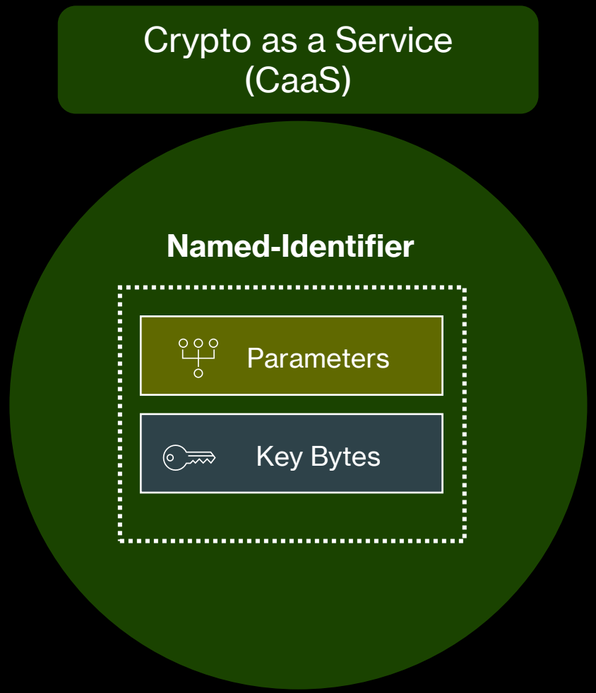
Input Abstraction

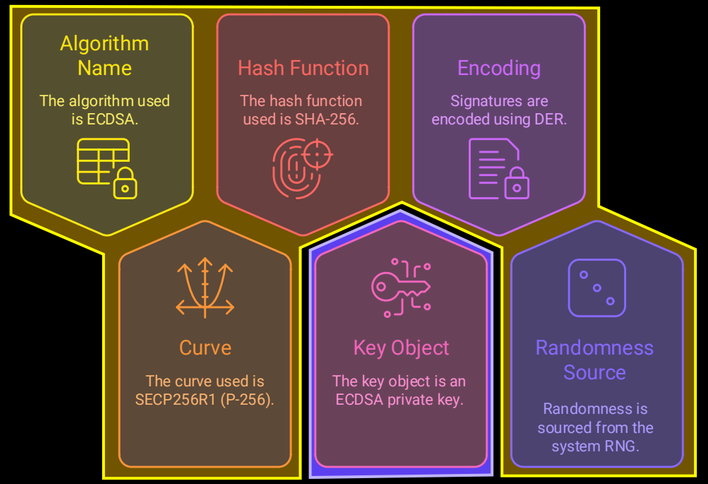


Input Abstraction



Input Abstraction

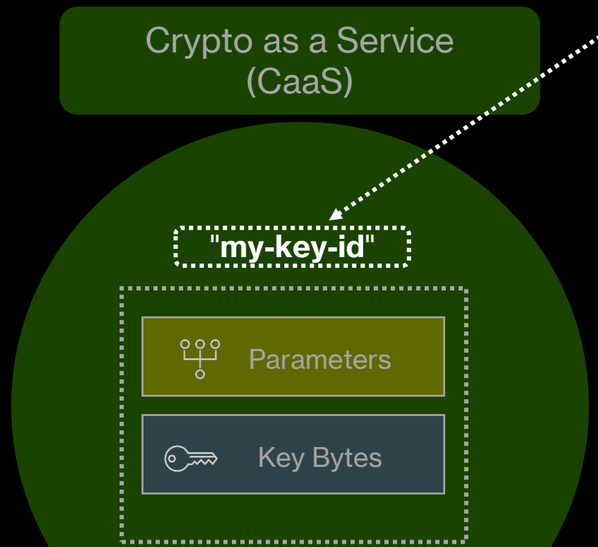




Hide key and parameters under a "*named-identifier*"

What If Signing Looked Like This?

```
# Old API: many knobs & dials
# New API: just “operation” + “named key identifier”
ciphertext = crypto_service.sign("my-key-id", b"data")
```



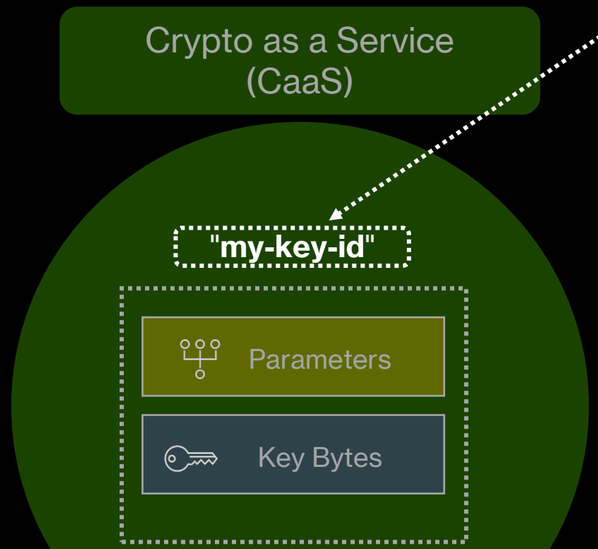
Under the hood

my-key-id → resolves to a key with certain set of parameters

Decouples app from Crypto Specifics

What If Signing Looked Like This?

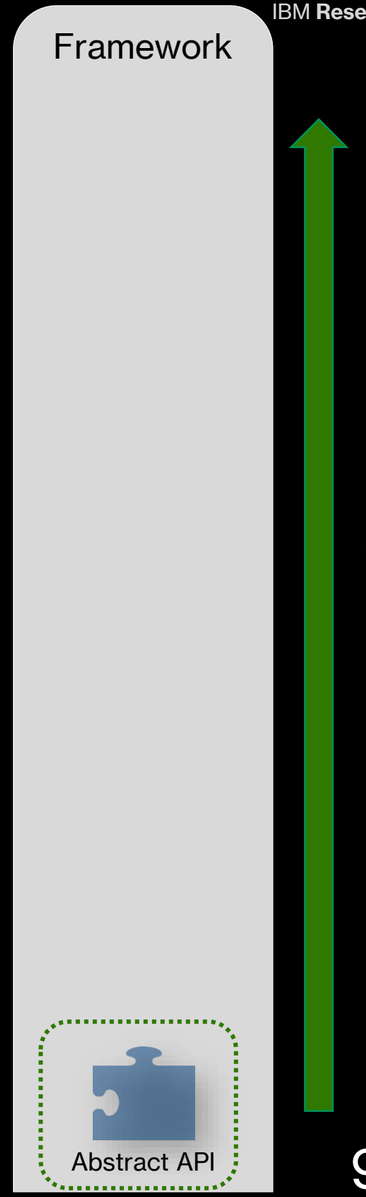
```
# Old API: many knobs & dials
# New API: just “operation” + “named key identifier”
ciphertext = crypto_service.sign("my-key-id", b"data")
```



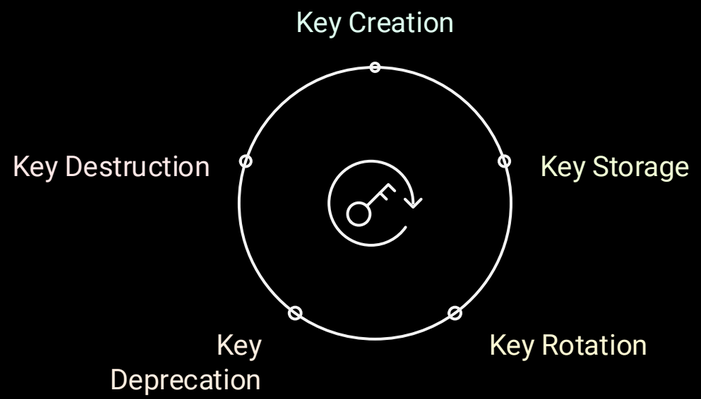
Under the hood

my-key-id → resolves to a key with certain set of parameters

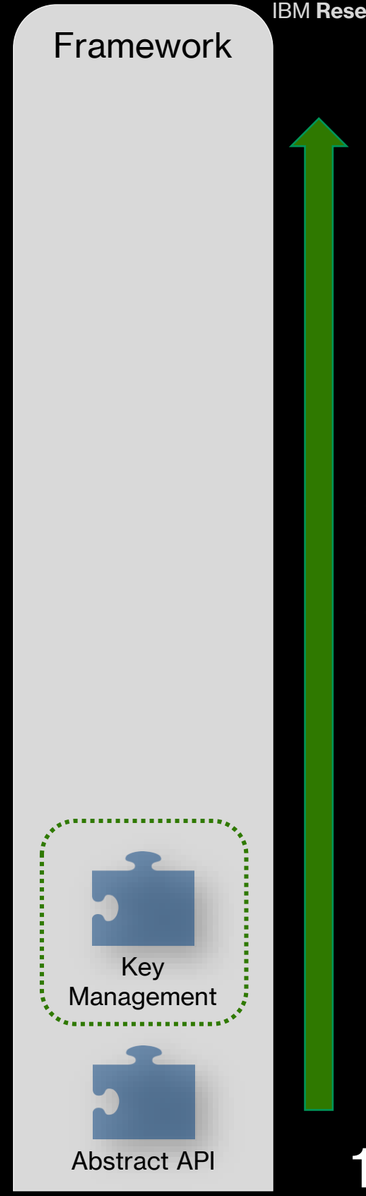
`Decouples app from Crypto Specifics`



Key Lifecycle Management

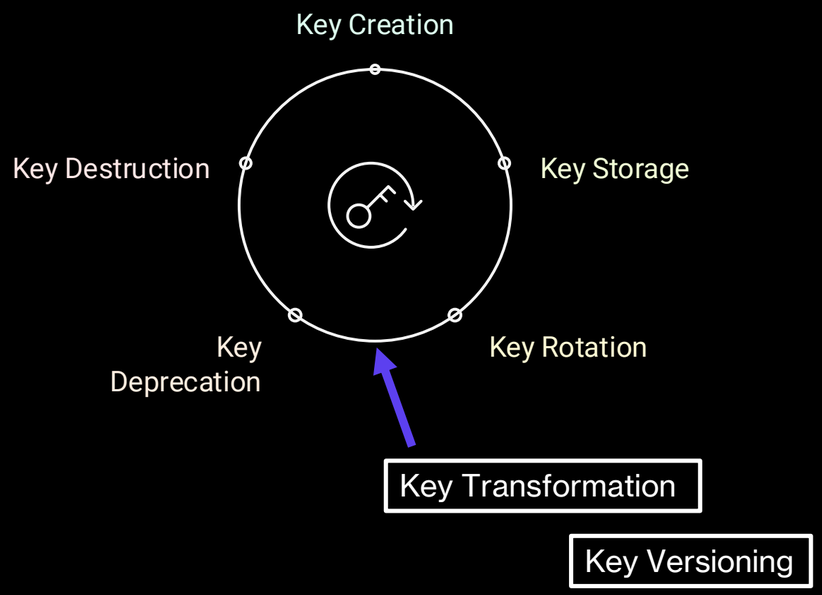


If keys are abstract, their lifecycle must be managed



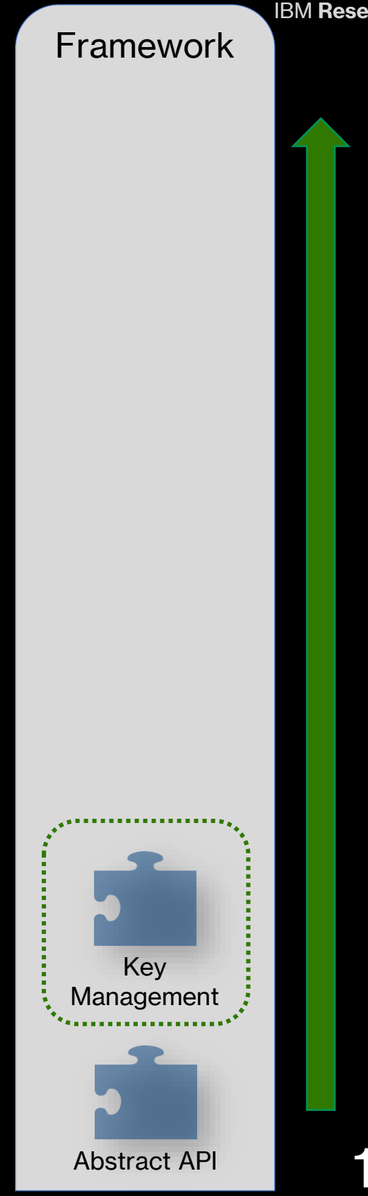
Key Lifecycle Management

Lifecycle Management ("my-key-id")

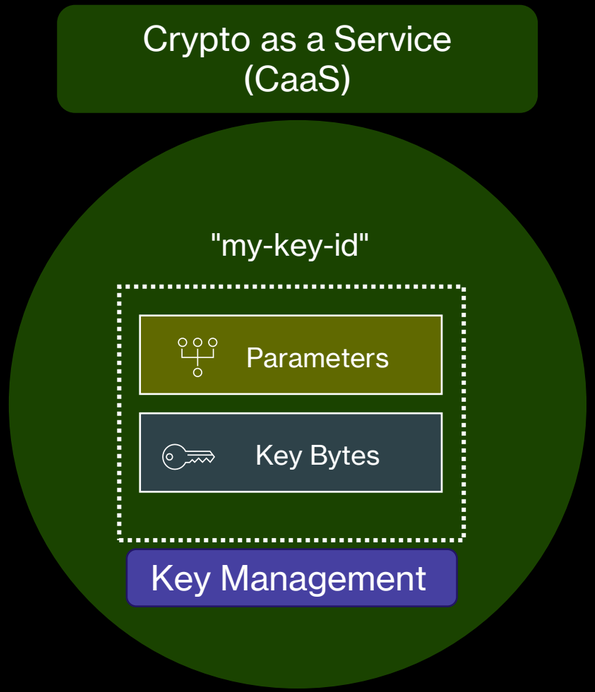


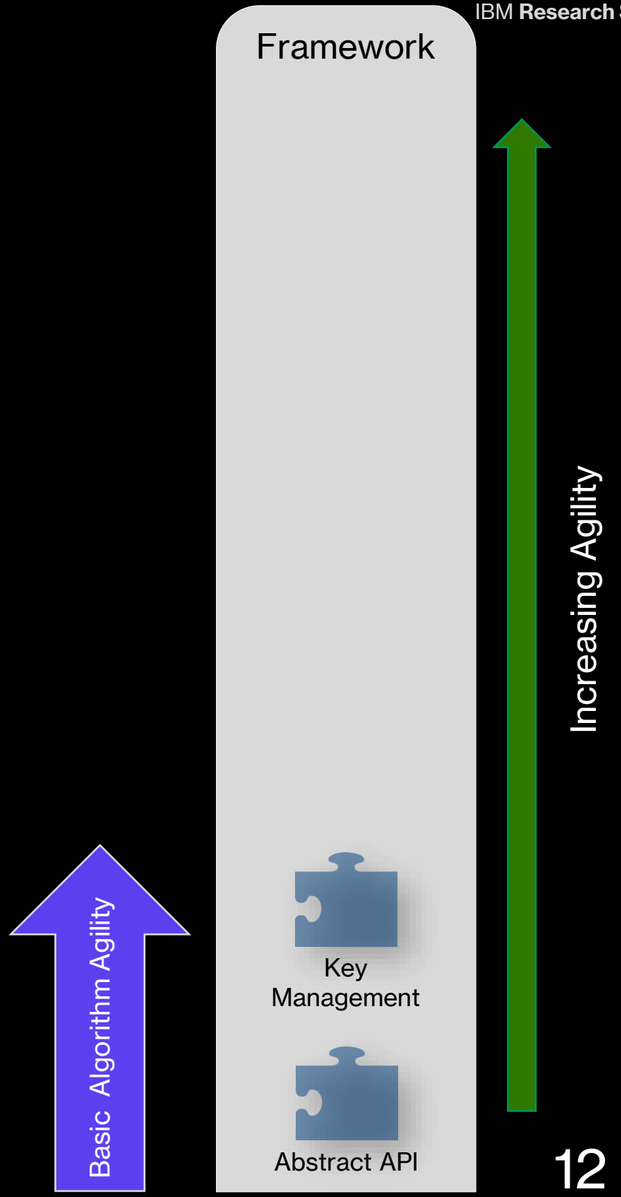
If keys are abstract, their lifecycle must be managed

Action	Key Bytes	Parameters
Rotation	Changes	Retained
Transformation	Can-Change	Can-Change



What Have We Achieved So Far?





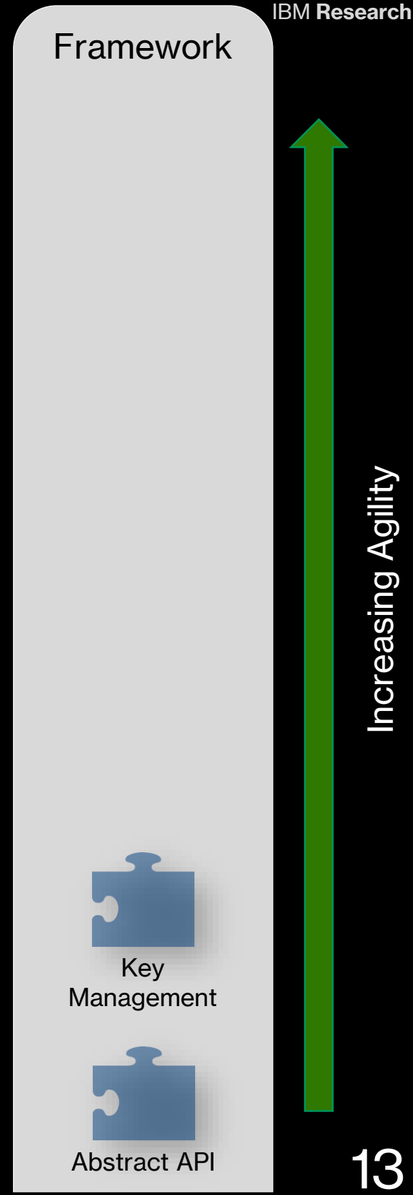
How To Create The Key?

ECDSA Signature

```
# Old API: many knobs & dials

# Create key with a template that captures relevant parameters
crypto_service.create_key("my-key-id", "ECDSA_P256_SHA256")

# New API: just “operation” + “named key identifier”
ciphertext = crypto_service.sign("my-key-id", b"data")
```

Use a key template that captures all the relevant parameters

We still need to know the key type!!



Replace "Template" with "Scope"

ECDSA Signature

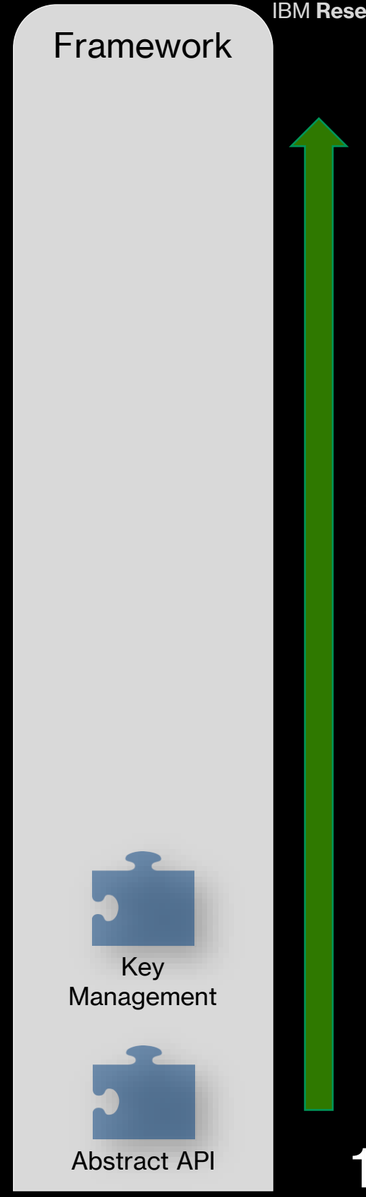
```
# Old API: many knobs & dials

# Create key with a template that captures relevant parameters
crypto_service.create_key("my-key-id", "ECDSA_P256_SHA256")

# New API: just “operation” + “named key identifier”
ciphertext = crypto_service.sign("my-key-id", b"data")
```

ECDSA Signature

```
# Old API: many knobs & dials

# Create key with a scope for intended use
crypto_service.create_key("my-key-id", "signature")

# New API: just “operation” + “named key identifier”
ciphertext = crypto_service.sign("my-key-id", b"data")
```

Scope: Intended purpose of use

- Authenticated data encryption
- Block Cipher
- Signature
- MAC/HMAC ...



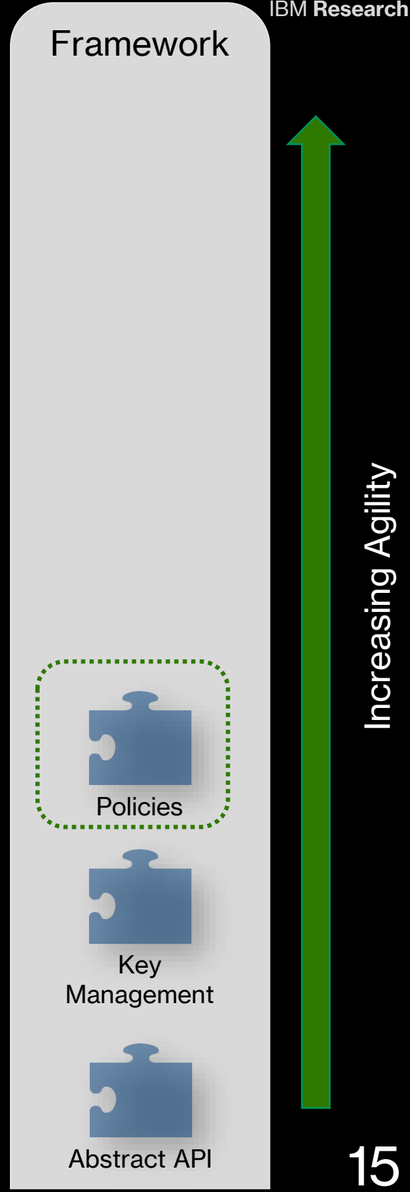
Policy Auto-Selects Algorithm

Scope	Preferred Algorithm	Fallback Algorithm
Authenticated Data Encryption	AES-GCM-96	ChaCha20-Poly1305
Signature	ECDSA-P256-SHA256	Ed25519

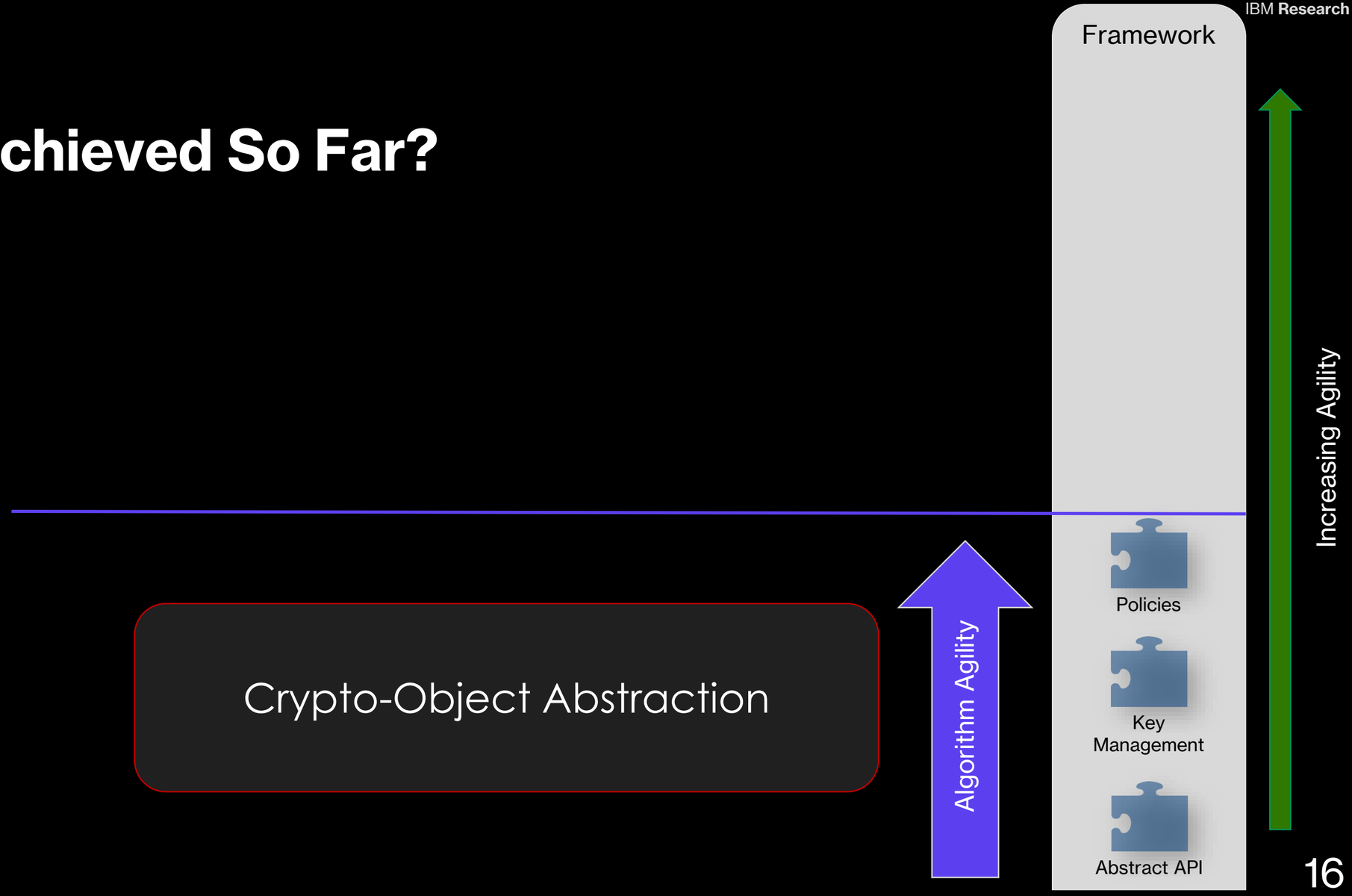
Under the hood

Policies determine which *algorithms* are allowed for each *scope*

Changing policies swaps algorithms without touching app code

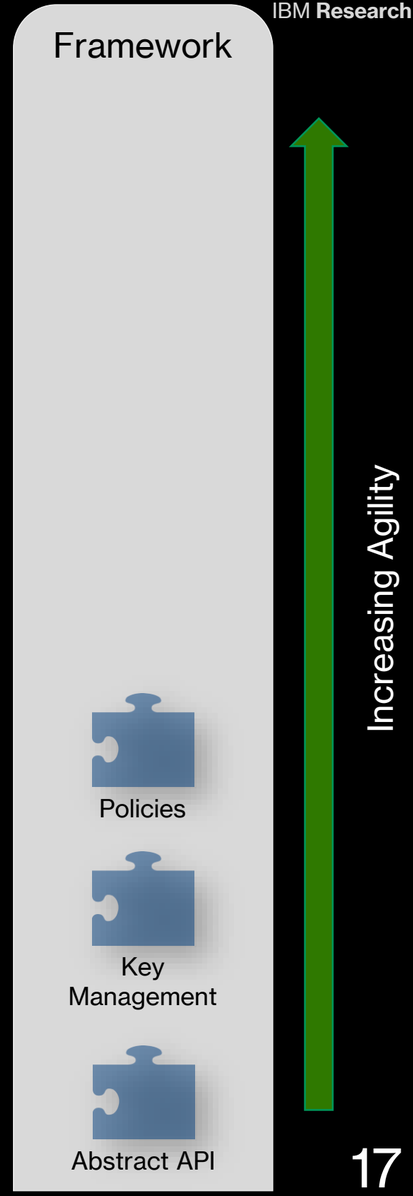


What Have We Achieved So Far?



Who Sets the Policies and Rules?

Application developers should not be able to set policies

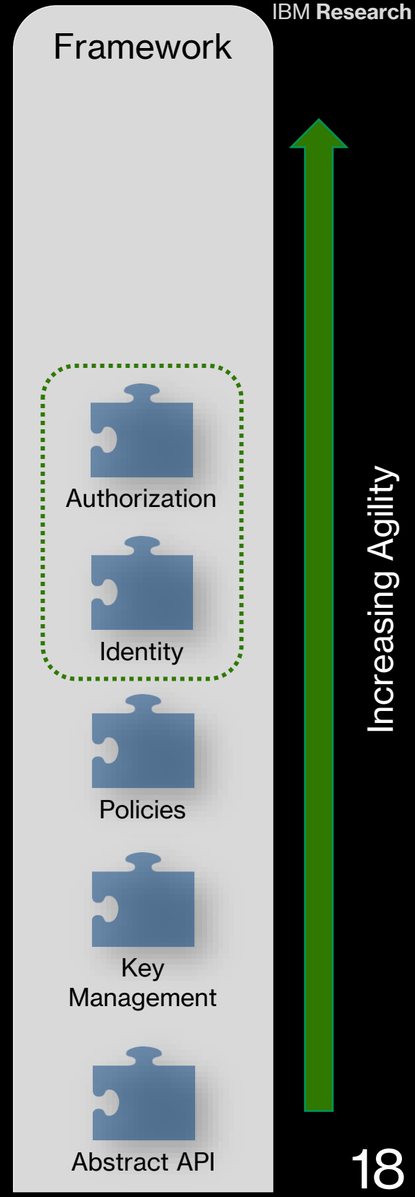


Who Sets the Policies and Rules?

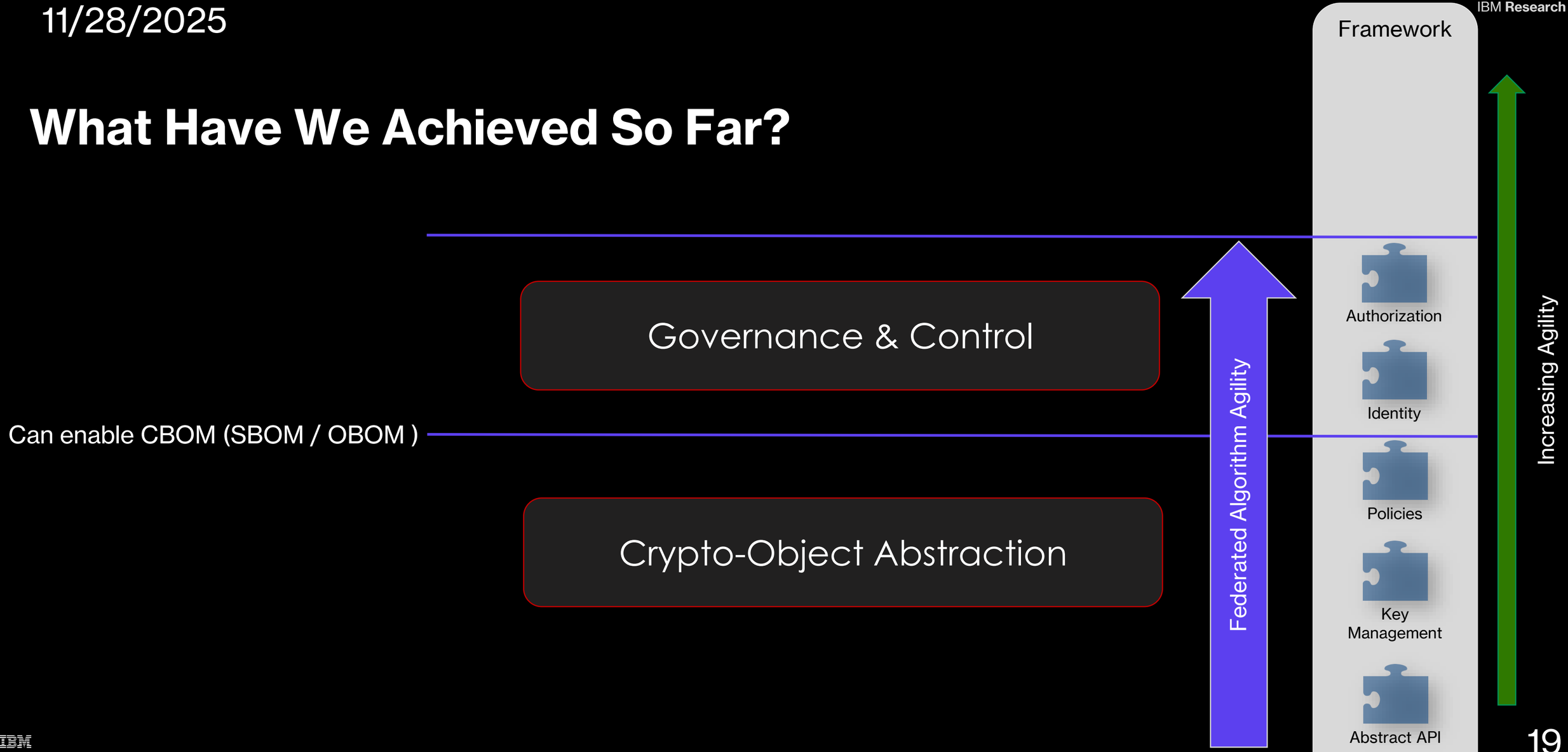
Application developers should not be able to set policies

Identity and Access Management for separation of responsibilities

- CISO/Crypto Admin → Sets Policies
- App Developer → Uses crypto via key IDs



What Have We Achieved So Far?



What Have We Achieved So Far?

Processing Agility

Governance & Control

Can enable CBOM (SBOM / OBOM)

Crypto-Object Abstraction

Crypto-Workflow Agility

Framework

Authorization

Identity

Policies

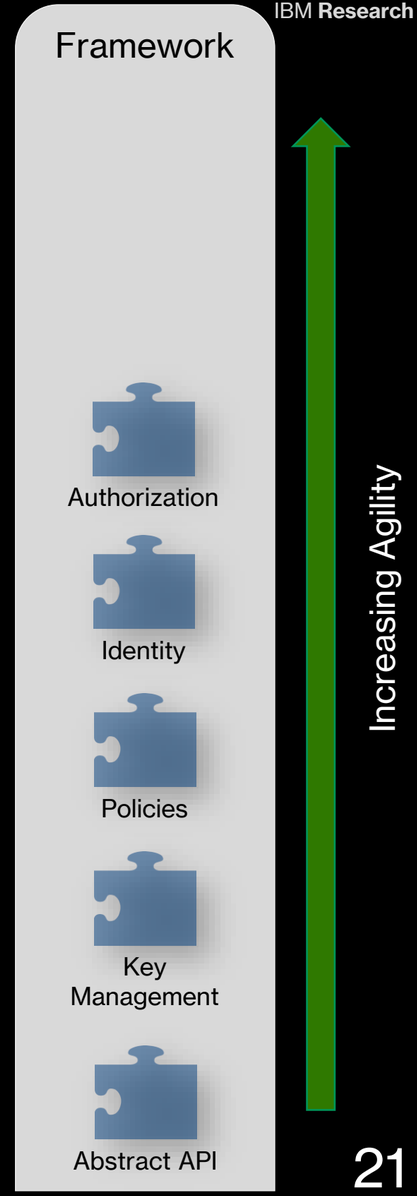
Key Management

Abstract API

Increasing Agility

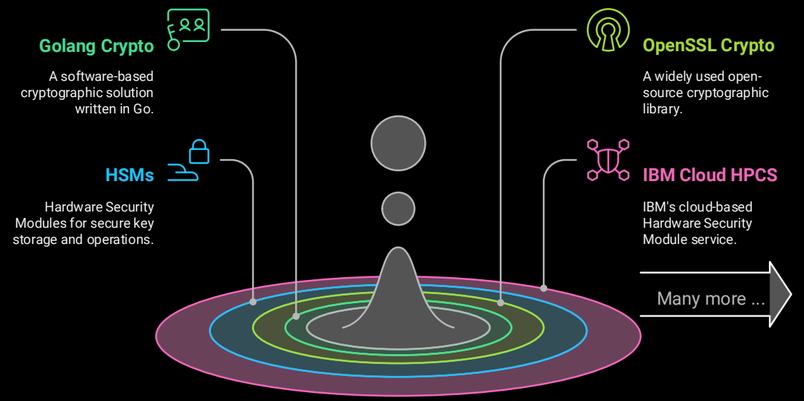
Crypto-Processing Agility

Where is Cryptography executed?



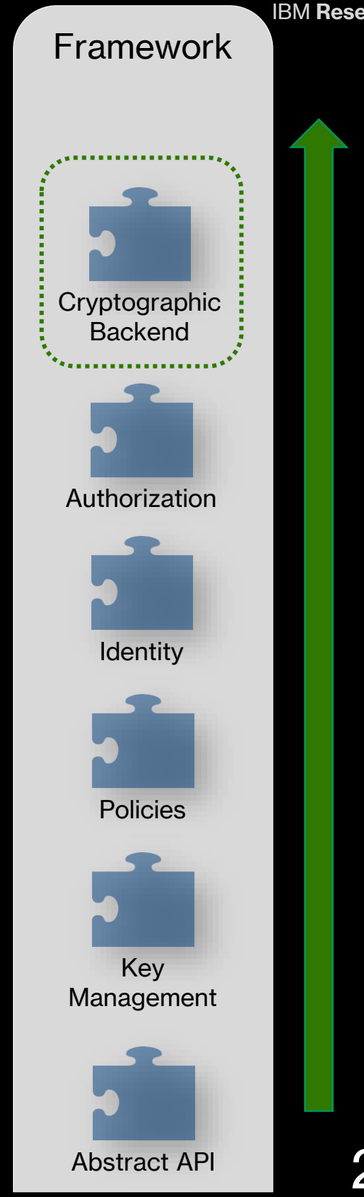
Crypto-Processing Agility

Where is Cryptography executed?

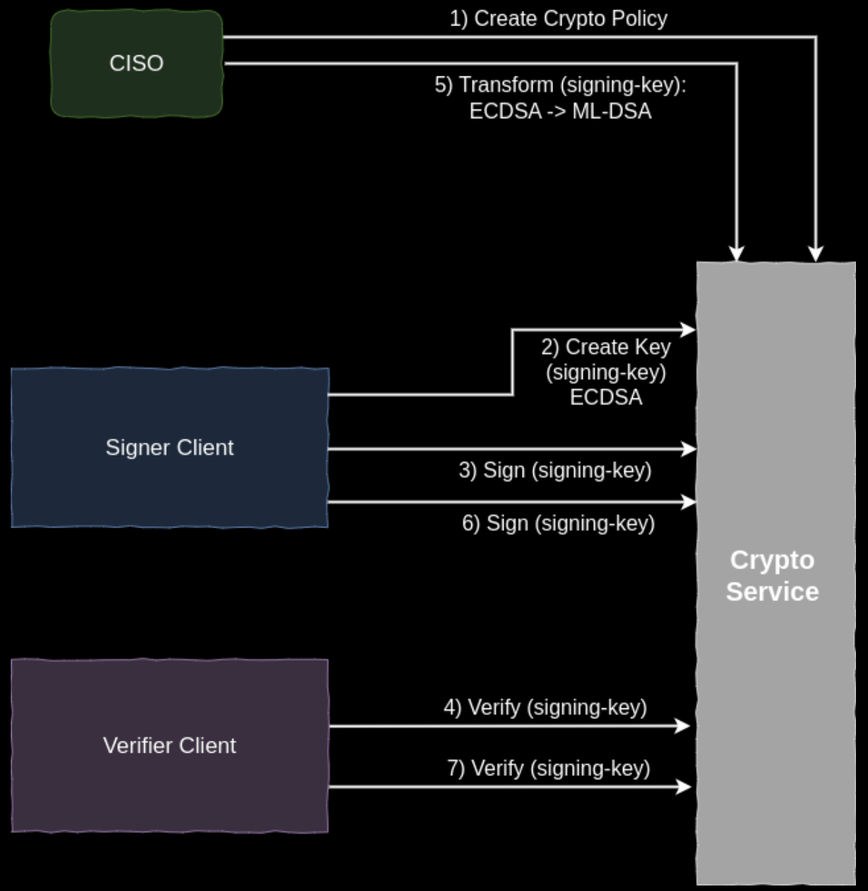


Backend = Crypto Libraries, HSM, Cloud KMS …

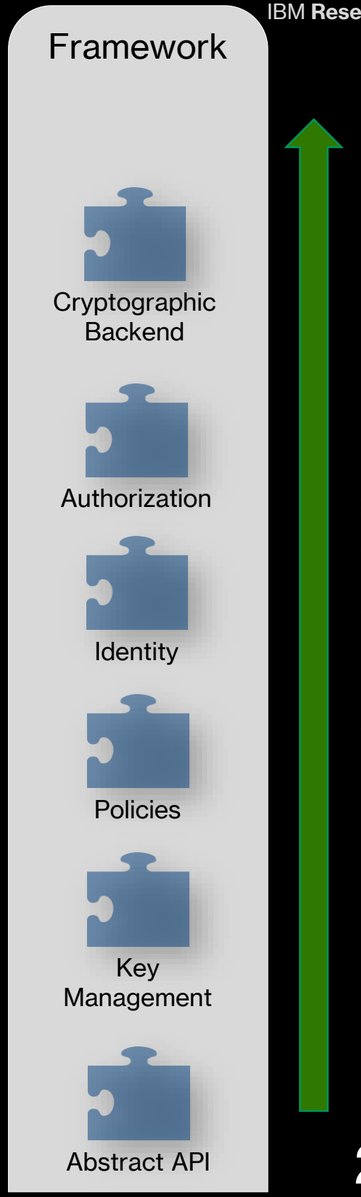
Abstract API and policy lets us switch cryptographic implementations at will



Sample Agile Workflow

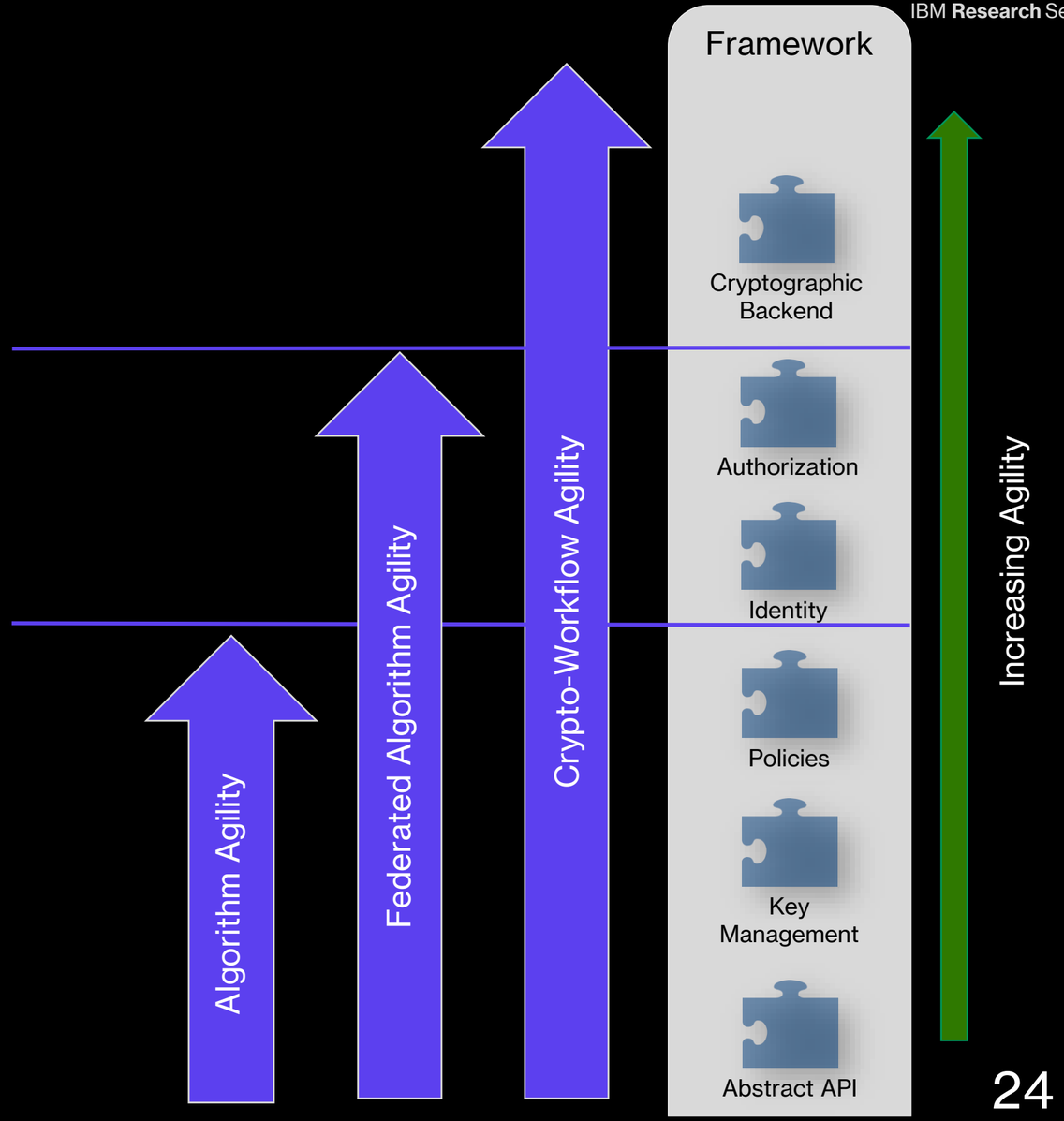


1. CISO officer creates a crypto policy which allows both ECDSA and ML-DSA algorithms and keys
2. Signer client creates an ECDSA Key called (“signing-key")
3. Signer client signs with key “signing-key”, and gets back a signature
4. Verifier client verifies the signature with key “signing-key”
5. The CISO officer now transforms the key signing-key to ML-DSA or updates policy to use ML-DSA
6. The signer client signs a new transaction with key “signing-key”. Under the hood, without any application-level code changes, this automatically is signed with ML-DSA
7. The verifier client verifies the new signature with key “signing-key” and gets a successful validation, again without any code changes



Adoption Path For Organizations

1. Replace direct crypto calls with abstract API
2. Centralize key lifecycle management
3. Introduce scope-based policies
4. Add strict identity & access control
5. Enable crypto-backend flexibility
6. ...



Putting It All Together

Thank You

Questions?