Commons Security

Cryptography made simple

Avec112

Version 1.0-SNAPSHOT, 2025-11-03

Table of Contents

Overview . . . 2
- Some common classes . . . 3

Asymmetric Encryption . . . 3
- RSA Encryption . . . 3
- ECC (Elliptic Curve Cryptography) . . . 5

Symmetric Encryption . . . 6
- AES Encryption . . . 6

Hybrid Encryption . . . 7
- Hybrid Encryption and Decryption . . . 7
- JSON Serialization . . . 8
- Human-Readable Description . . . 8

Shamir's Secret Sharing . . . 9
- Shamir example . . . 10

Digest (hashing) . . . 10
- Example . . . 11

Signature and verification . . . 11
- RSA Signatures (RSASSA-PSS) . . . 11
- Ed25519 Signatures (ECC-based) . . . 12
- ECDSA Signatures (ECC-based) . . . 13

Key Storage and Management . . . 13
- Storing and Loading RSA Keys . . . 13
- Storing and Loading ECC Keys . . . 14
- Storing and Loading AES Keys . . . 14
- Key Format Conversion . . . 15
- Key Validation . . . 16

Password Encoder . . . 16
- Encoding . . . 16
- Matching . . . 17
- Upgrading Passwords . . . 17

Documentation . . . 18
- PDF Version . . . 18
- Generate PDF locally . . . 19

Recommended reading . . . 19

TODO . . . 19

Educational and Internal Use Only

This library is intended for **educational and internal use only**.

It is **not intended for production deployment** or active maintenance as a public library. The codebase serves as a reference implementation and learning resource for cryptographic concepts and best practices in Java.

For production systems, please use well-established and actively maintained security libraries such as:

- Spring Security
- BouncyCastle
- Google Tink

Overview

Commons Security is a Java library with several encryption features for `Java 11` and higher build on top of *Java Cryptography Architecture* (JCA)[1] and *Java Cryotography Extension* (JCE)[2] with *Bouncy Castle*[3] as provider. The library hides the implementation behind a simple and easy to use API. Typically a simplified API requires that some decisions has been made already, like which algorithm to use, encryption strength, etc. However do not worry, look behind the simplified API and you will find more configuration options.

Features

- **Asymmetric encryption** and decryption with *public key* and *private key*. `RSA` and `ECIES` (ECC-based) are supported.
- **Symmetric encryption** and decryption with one (same) key. `AES/CTR` and `AES/GCM` is supported.
- **Hybrid encryption** combines RSA and AES for secure and efficient encryption of large data. RSA encrypts the AES key, while AES encrypts the actual data.
- **Shamir's Secret Sharing** is a way to split a secret into shares. The secret can be recreated by putting the minimum required amount of shares back together.
- **Digest** or message digest is the result of hashing and is a way to integrity check our data. A typical use case could be to compare content (hashed) with a stored hash to verify if the data has been altered.
- Digital **Signature** and **verification** using RSA keypair with modern `RSASSA-PSS` padding, or ECC with `Ed25519` (modern, fast) or `ECDSA` (standards-compliant). The user signs data with their *private key* and others verify the signature with the originator's *public key*.
- **Password Encoder** uses proven algorithms for hashing passwords in a safe and secure way.

You may use the simple delegate class `CryptoUtil` or you can access classes like `RsaCipher`, `AesCipher`, `Shamir`, etc. directly. The latter might give you more options. Implementation alternatives are shown in examples below.

The `CryptoUtil` facade provides convenient methods for:

- **RSA encryption/decryption** - `rsaEncrypt()` / `rsaDecrypt()`
- **ECIES encryption/decryption** - `eciesEncrypt()` / `eciesDecrypt()` (ECC-based, smaller keys than RSA)
- **AES encryption/decryption** - `aesEncrypt()` / `aesDecrypt()`
- **Hybrid encryption/decryption** - `hybridEncrypt()` / `hybridDecrypt()`
- **Shamir's Secret Sharing** - `getShamirShares()` / `getShamirSecret()`
- **Digest (hashing)** - `digest()` / `base64Digest()` / `hexDigest()`
- **Digital signatures** - `sign()` / `verify()` (RSA), `signEd25519()` / `verifyEd25519()` (ECC), `signEcdsa()` / `verifyEcdsa()` (ECC)
- **Password encoding** - `encodePassword()` / `matchesPassword()` / `needsPasswordUpgrade()` / `upgradePassword()`

Some common classes

This is some common classes used when working with *Symmetric* and *Asymmetric encryption* but also *Password Encoding*.

- `PlainText` is a placeholder for unencrypted text.
  `PlainText plainText = new PlainText("My secret plaintext");`
- `CipherText` is a placeholder for encrypted text.
  `CipherText cipherText = new CipherText("ymEIVhbBPhWAIzDx7MalbeLoccwnw=");`
- `Password` is a placeholder for a string (the password) used for either encryption or decryption.
  `Password password = new Password("SecretPassword123!");`
- `PublicKey` holds the *public key* derived from `java.security.KeyPair` and can be used for encryption or signature verification.
  `PublicKey publicKey = keyPair.getPublicKey();`
- `PrivateKey` holds the *private key* derived from `java.security.KeyPair` and can be used for decryption or for creating a signature.
  `PrivateKey privateKey = keyPair.getPrivateKey();`

Asymmetric Encryption

Asymmetric encryption uses a *public key* for encryption and a *private key* for decryption. This library supports both RSA and ECC-based encryption.

RSA Encryption

RSA[4] (Rivest–Shamir–Adleman) is a *public-key* cryptosystem that is widely used for secure data transmission. In a *public-key* cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private).

> It involves the use of two mathematically related keys. The public key (the

> one that's known to everybody) and the private key (which is only known by you) are required for encrypting and decrypting the message. The public key can be derived from the private key but not the other way.

Asymmetric encryption use a *public key* for encryption and a *private* key for decryption. Useful when you want to share secrets with others. You may encrypt a file with someone's *public key*, so they will be able to decrypt with their matching *private key*.

> Used to share small files or text with other users. For large content (more than 117 bytes) use AES symmetric encryption. The AES secret key can later be shared by using *Public Key Encryption* like RSA encryption or *Diffie-Hellman*[5].

RSA KeyPair

RSA requires a *private key* and a matching *public key*. Supported key sizes (encryption strength) are `2048`, `3072` (default) and `4096`.

> A key size of `3072` bit or higher is recommended for strong security. Key size `1024` is not supported.

> Any secret encrypted with a *public key* can only be decrypted with the associated *private key*.

Using `KeyGeneratorUtil` *to generate a* `4096` *bit* `KeyPair`

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair4096();
```

Alternative

```
RsaCipher rsaCipher = new RsaCipher();
KeyPair keyPair = rsaCipher.generateKeyPair(KeySize.BIT_4096);
```

RSA Encryption and Decryption

RSA encryption and decryption using `CryptoUtil`

```
KeyPair keyPair = CryptoUtil.generateRsaKeyPair4096();
CipherText cipherText = CryptoUtil.rsaEncrypt("My secret", keyPair.getPublicKey());
PlainText plainText = CryptoUtil.rsaDecrypt(cipherText, keyPair.getPrivateKey());
```

Alternative

```
RsaCipher rsaCipher = new RsaCipher();
KeyPair keyPair = rsaCipher.generateKeyPair(KeySize.BIT_4096);
CipherText cipherText = rsaCipher.encrypt("My secret", keyPair.getPublicKey());
```

```
PlainText plainText = rsaCipher.decrypt(cipherText, keyPair.getPrivateKey());
```

ECC (Elliptic Curve Cryptography)

ECC provides equivalent security to RSA with much smaller key sizes, resulting in faster operations and reduced memory footprint.

Key Size Comparison

- 256-bit ECC ≈ 3072-bit RSA
- 384-bit ECC ≈ 7680-bit RSA
- 521-bit ECC ≈ 15360-bit RSA

Supported ECC Curves

- **Ed25519** - Modern signature curve (256-bit security)
- **secp256r1 (P-256)** - NIST curve for ECDSA and ECIES (256-bit security)
- **secp384r1 (P-384)** - NIST curve for higher security (384-bit security)
- **secp521r1 (P-521)** - NIST curve for maximum security (521-bit security)

ECC Key Generation

Generate Ed25519 keypair for signatures

```
KeyPair ed25519Keys = KeyGeneratorUtil.generateEd25519KeyPair();
```

Generate secp256r1 keypair for ECDSA or ECIES

```
KeyPair ecKeys = KeyGeneratorUtil.generateSecp256r1KeyPair();
```

Generate with specific curve

```
KeyPair ecKeys = KeyGeneratorUtil.generateEcKeyPair(EccCurve.SECP384R1);
```

ECIES Encryption

ECIES (Elliptic Curve Integrated Encryption Scheme) is a hybrid encryption scheme that combines ECC key agreement with symmetric encryption. It's the ECC equivalent of RSA-OAEP.

ECIES encryption using `CryptoUtil`

```
KeyPair keyPair = KeyGeneratorUtil.generateSecp256r1KeyPair();

// Encrypt
byte[] ciphertext = CryptoUtil.eciesEncrypt("Secret message", keyPair.getPublic());
```

```
// Decrypt
String plaintext = CryptoUtil.eciesDecrypt(ciphertext, keyPair.getPrivate());
```

Alternative using `EciesCipher` *directly*

```
KeyPair keyPair = KeyGeneratorUtil.generateSecp256r1KeyPair();

byte[] ciphertext = EciesCipher.encrypt("Secret message", keyPair.getPublic());
String plaintext = EciesCipher.decrypt(ciphertext, keyPair.getPrivate());
```

> ECIES provides built-in integrity protection (MAC) and uses smaller keys than RSA for the same security level.

Symmetric Encryption

Symmetric encryption uses the same key for both encryption and decryption. This library supports AES encryption.

AES Encryption

The Advanced Encryption Standard (AES)[6], also known by its original name Rijndael is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

> There's a single shared key that's used for encryption and decryption.

> Useful for encryption larger files compared to asymmetric encryption. A symmetric key can be shared (distributed) by using *RSA public key encryption*, *Diffie-Hellman* or even *Sharmir's Secret Sharing* if needed.

Modes supported: `GCM` (default), `CTR`
Strength supported: `128`, `192`, `256` (default)

AES Encryption and Decryption

AES encryption and decryption using `CryptoUtil` *with* `GCM`*@*`256`*-bit*

```
Password password = new Password("SecretPassword123!");
CipherText cipherText = CryptoUtil.aesEncrypt("Secret text", password);
PlainText plainText = CryptoUtil.aesDecrypt(cipherText, password);
```

Alternative (here as `CTR`*@*`192`*-bit)*

```
Password password = new Password("SecretPassword123!");
AesCipher aesCipher = new AesCipher(EncryptionMode.CTR, EncryptionStrength.BIT_192);
```

```
h cipherText = aesCipher.encrypt("Secret text", password);
PlainText plainText = aesCipher.decrypt(cipherText, password);
```

Hybrid Encryption

Hybrid encryption combines the strengths of both asymmetric (RSA) and symmetric (AES) encryption. RSA is used to securely encrypt a randomly generated AES key, while AES encrypts the actual data payload. This approach provides the security of RSA with the performance and capacity of AES.

> RSA can only encrypt small amounts of data (typically < 200 bytes depending on key size and padding). Hybrid encryption solves this by using RSA to encrypt a symmetric key, which then encrypts arbitrarily large data.

Hybrid encryption is the recommended approach for encrypting large files or data. It's the same technique used in protocols like TLS/SSL and PGP.

Hybrid Encryption and Decryption

Hybrid encryption and decryption using `CryptoUtil`

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair3072();
PlainText plainText = new PlainText("Large amount of sensitive data...");

// Encrypt
HybridEncryptionResult result = CryptoUtil.hybridEncrypt(plainText, keyPair.
getPublic());

// Decrypt
PlainText decrypted = CryptoUtil.hybridDecrypt(result, keyPair.getPrivate());
```

Alternative using builders with custom AES mode and strength

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair3072();
PlainText plainText = new PlainText("Large amount of sensitive data...");

// Encrypt with custom settings
HybridEncryptionResult result = EncryptBuilder.encryptionBuilder()
    .plainText(plainText)
    .key(keyPair.getPublic())
    .withMode(EncryptionMode.GCM)
    .withStrength(EncryptionStrength.BIT_256)
    .build();

// Decrypt
PlainText decrypted = DecryptBuilder.decryptionBuilder()
```

```
    .key(keyPair.getPrivate())
    .cipherText(result.getCipherText())
    .encryptedKey(result.getEncryptedKey())
    .withMode(result.getAesEncryptionMode())
    .withStrength(result.getAesEncryptionStrength())
    .build();
```

The `HybridEncryptionResult` contains the AES-encrypted data, the RSA-encrypted AES key, and metadata about the encryption mode and strength used.

JSON Serialization

`HybridEncryptionResult` supports JSON serialization for easy storage and transmission:

Serialize to JSON and deserialize back

```
// Encrypt
HybridEncryptionResult result = CryptoUtil.hybridEncrypt(plainText, keyPair.
getPublic());

// Serialize to JSON (for storage or transmission)
String json = result.toJson();

// Later... deserialize from JSON
HybridEncryptionResult restored = HybridEncryptionResult.fromJson(json);

// Decrypt using restored result
PlainText decrypted = CryptoUtil.hybridDecrypt(restored, keyPair.getPrivate());
```

The JSON format includes a version field for future compatibility:

```
{
  "version": "1.0",
  "cipherText": {
    "value": "encrypted-data-here"
  },
  "encryptedKey": "rsa-encrypted-key-here",
  "aesEncryptionMode": "GCM",
  "aesEncryptionStrength": "BIT_256"
}
```

Human-Readable Description

The `describe()` method provides a convenient way to get a human-readable summary of the encryption configuration:

Get encryption configuration description

```
HybridEncryptionResult result = CryptoUtil.hybridEncrypt(plainText, keyPair.
getPublic());

// Get human-readable description
String description = result.describe(); // Returns "GCM@256-bit"

// Useful for logging
log.info("Data encrypted using: {}", result.describe());
```

Shamir's Secret Sharing

Shamir's Secret Sharing[7] is used to share a secret in a distributed way, most often to secure other encryption keys. These shares are used to reconstruct the original secret.

To unlock the secret via Shamir's Secret Sharing, a minimum number of shares are needed. This is called the threshold, and is used to denote the minimum number of shares needed to unlock the secret.

This implementation is based on the work of *Coda Hale*[8] and his project *Shamir*[9].

Shamir's Secret Sharing using `CryptoUtil`

```
int totalShares = 5;
int minShares = 2; // threshold
Secret secret = new Secret("Secret text");

// Split the secret into 5 shares
Shares shares = CryptoUtil.getShamirShares(secret, totalShares, minShares);
Share bob = shares.get(0);
Share alice = shares.get(1);

// Reconstruct the secret using any combination of at least `minShares` shares
Secret reconstructed = CryptoUtil.getShamirSecret(bob, alice);
```

The API validates invalid configurations such as `threshold < 2` or `threshold > totalShares`.

Alternative

```
int totalShares = 5;
int minShares = 2; // threshold
Secret secret = new Secret("Secret text");

// Split the secret into 5 shares
Shares shares = Shamir.getShares(secret, totalShares, minShares);
Share bob = shares.get(0);
```

```
Share alice = shares.get(1);

// Reconstruct the secret using any combination of at least `minShares` shares
Secret reconstructed = Shamir.getSecret(bob, alice);
```

Shamir example

A bank has a vault full of money. The bank's policy requires that nobody should be able to open the vault alone. Five employees are selected to have access to the vault and there must be at least two (2) employees at any time when opening the vault.

- **Split phase:** Five (5) keys are being distributed. `Bob`, `Alice`, `Eve`, `Tom` and `Lisa` all get one *share* each using *Shamir's Secret Sharing* to split the secret into five shares.
- **Join phase:** It's time to open the safe. The requirement is two (2) shares to open the vault. `Bob` and `Alice` bring their shares. By using *Shamir's Secret Sharing* the shares from both will be joined and the secret recreated.

Digest (hashing)

Digest or *message digest* is the result of *hashing*[10] data or content. The hashing is a one-way compression function to convert inputs of different lengths into a fixed-length output (hash value). The default used by this library is `SHA-2@SHA-512/256`. It provides 256-bit output with the internal 64-bit strength of SHA-512, combining high security with good performance on modern CPUs. Up to `SHA-2@SHA-512` is possible with `JDK 8`. Stronger and newer hashing like `SHA3` is not supported out of the box with `JDK 8`. That would require `JDK 9` or higher. However *BouncyCastle*[11] has been added as main provider and `SHA-3` is therefore available.

> SHA-512/256 offers the same collision resistance as SHA-512, but produces a shorter digest (256 bits) and is faster on 64-bit systems. Other algorithms such as SHA3-256 and SHA3-512 are also supported when using the BouncyCastle provider.

> Do not use Message Digest for password storage. For that you should use KDF[12] algorithms like `ARGON2`, `BCRYPT`, `SCRYPT` or `PBKDF2`. Se *Password Encoder* below.

Non supported hashing algorithms

The following hashing algorithms are **not** recommended or supported: `MD4`, `MD5`, `SHA-0` and `SHA-1`

Supported hashing algorithms

```
    SHA_256("SHA-256"),
    SHA_512_256("SHA-512/256"), // default
    SHA3_256("SHA3-256"),
    SHA3_512("SHA3-512");
```

Worth reading

- Read more about *Secure Hash Algorithms*[13]
- Nice article about *hashing security*[14]

Example

Using `CryptoUtil` *(recommended for simple use cases)*

```
// Raw bytes (SHA-512/256)
final byte[] digest = CryptoUtil.digest("My data");

// Base64 encoded
final String base64Digest = CryptoUtil.base64Digest("My data");

// Hex encoded
final String hexDigest = CryptoUtil.hexDigest("My data");
```

Alternative using `DigestUtil` *directly*

```
// Raw bytes
final byte[] digest = DigestUtil.digest(data);

// Base64 encoded
final String digest = DigestUtil.base64Digest(data);

// Hex encoded
final String digest = DigestUtil.hexDigest(data);

// Other hashing algorithms (example SHA3-256)
final byte[] digest = DigestUtil.digest(data, DigestAlgorithm.SHA3_256);
```

Signature and verification

A *digital signature*[15] is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very strong reason to believe that the message was created by a known sender (authentication), and that the message was not altered in transit.

This library supports multiple signature algorithms:

- **RSASSA-PSS** - Modern RSA signature scheme with probabilistic padding
- **Ed25519** - Modern ECC signature (fastest, deterministic)
- **ECDSA** - Standards-compliant ECC signature (probabilistic)

RSA Signatures (RSASSA-PSS)

Commons Security uses `RSASSA-PSS` (SHA-256 with MGF1)[16] — the modern and recommended RSA

signature scheme replacing legacy `SHA256withRSA`.

Using `CryptoUtil` (recommended for simple use cases)

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair2048();

// Sign data
byte[] signature = CryptoUtil.sign("My data", keyPair.getPrivate());

// Verify signature
boolean verified = CryptoUtil.verify(signature, "My data", keyPair.getPublic());
```

Alternative using `SignatureUtil` directly

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair2048();
byte[] signature = SignatureUtil.sign("My data", keyPair.getPrivate());
boolean verified = SignatureUtil.verify(signature, "My data", keyPair.getPublic());
```

RSASSA-PSS provides probabilistic padding, making it more secure than the traditional PKCS#1 v1.5 (`SHA256withRSA`).

Ed25519 Signatures (ECC-based)

Ed25519 is a modern signature algorithm providing:

- **High performance** - Much faster than RSA and ECDSA
- **Strong security** - 128-bit security (equivalent to RSA-3072)
- **Deterministic** - Same input always produces same signature
- **Small signatures** - Only 64 bytes per signature
- **Small keys** - 32-byte public keys, 32-byte private keys

Ed25519 signatures using `CryptoUtil`

```
KeyPair keyPair = KeyGeneratorUtil.generateEd25519KeyPair();

// Sign data
byte[] signature = CryptoUtil.signEd25519("My data", keyPair.getPrivate());

// Verify signature
boolean verified = CryptoUtil.verifyEd25519(signature, "My data", keyPair.
getPublic());
```

Alternative using `SignatureUtil` directly

```
KeyPair keyPair = KeyGeneratorUtil.generateEd25519KeyPair();
byte[] signature = SignatureUtil.signEd25519("My data", keyPair.getPrivate());
```

```
boolean verified = SignatureUtil.verifyEd25519(signature, "My data", keyPair.
getPublic());
```

Ed25519 is used in modern protocols like SSH, Signal, and TLS 1.3.

ECDSA Signatures (ECC-based)

ECDSA provides standards-compliant ECC signatures with much smaller key sizes than RSA:

- 256-bit ECDSA ≈ 3072-bit RSA
- 384-bit ECDSA ≈ 7680-bit RSA
- 521-bit ECDSA ≈ 15360-bit RSA

ECDSA signatures using `CryptoUtil`

```
KeyPair keyPair = KeyGeneratorUtil.generateSecp256r1KeyPair();

// Sign data (hash algorithm auto-selected based on curve)
byte[] signature = CryptoUtil.signEcdsa("My data", keyPair.getPrivate());

// Verify signature
boolean verified = CryptoUtil.verifyEcdsa(signature, "My data", keyPair.getPublic());
```

Alternative using `SignatureUtil` *directly*

```
KeyPair keyPair = KeyGeneratorUtil.generateSecp256r1KeyPair();
byte[] signature = SignatureUtil.signEcdsa("My data", keyPair.getPrivate());
boolean verified = SignatureUtil.verifyEcdsa(signature, "My data", keyPair.
getPublic());
```

ECDSA signatures are probabilistic (include random nonce), meaning signing the same data twice produces different signatures.

Key Storage and Management

`KeyStorageUtil` provides utilities for storing and loading cryptographic keys in standard formats. It supports RSA, EC (ECDSA/ECIES), Ed25519, and symmetric AES keys.

Storing and Loading RSA Keys

Save and load RSA key pair

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair3072();

// Save keys to disk (PEM format)
```

```
KeyStorageUtil.savePrivateKey(keyPair.getPrivate(), Path.of("private.pem"));
KeyStorageUtil.savePublicKey(keyPair.getPublic(), Path.of("public.pem"));

// Load keys from disk
PrivateKey privateKey = KeyStorageUtil.loadPrivateKey(Path.of("private.pem"), "RSA");
PublicKey publicKey = KeyStorageUtil.loadPublicKey(Path.of("public.pem"), "RSA");
```

`savePrivateKey()` stores the key unencrypted. For production use, consider using encrypted storage (currently not implemented).

Storing and Loading ECC Keys

Save and load EC keys (secp256r1)

```
KeyPair keyPair = KeyGeneratorUtil.generateSecp256r1KeyPair();

// Save keys
KeyStorageUtil.savePrivateKey(keyPair.getPrivate(), Path.of("ec_private.pem"));
KeyStorageUtil.savePublicKey(keyPair.getPublic(), Path.of("ec_public.pem"));

// Load keys
PrivateKey privateKey = KeyStorageUtil.loadPrivateKey(Path.of("ec_private.pem"),
"EC");
PublicKey publicKey = KeyStorageUtil.loadPublicKey(Path.of("ec_public.pem"), "EC");
```

Save and load Ed25519 keys

```
KeyPair keyPair = KeyGeneratorUtil.generateEd25519KeyPair();

// Save keys
KeyStorageUtil.savePrivateKey(keyPair.getPrivate(), Path.of("ed25519_private.pem"));
KeyStorageUtil.savePublicKey(keyPair.getPublic(), Path.of("ed25519_public.pem"));

// Load keys
PrivateKey privateKey = KeyStorageUtil.loadPrivateKey(Path.of("ed25519_private.pem"),
"Ed25519");
PublicKey publicKey = KeyStorageUtil.loadPublicKey(Path.of("ed25519_public.pem"),
"Ed25519");
```

Storing and Loading AES Keys

Save and load AES key (unencrypted)

```
SecretKey secretKey = KeyGeneratorUtil.generateAesKey(AesKeySize.BIT_256);

// Save key (Base64 encoded)
KeyStorageUtil.saveAesKey(secretKey, Path.of("aes_key.txt"));
```

```
// Load key
SecretKey loadedKey = KeyStorageUtil.loadAesKey(Path.of("aes_key.txt"));
```

`saveAesKey()` stores the key in Base64 format without encryption. For production use, use `saveAesKeyEncrypted()`.

Save and load AES key (encrypted with password)

```
SecretKey secretKey = KeyGeneratorUtil.generateAesKey(AesKeySize.BIT_256);
Password password = new Password("StrongPassword123!");

// Save encrypted
KeyStorageUtil.saveAesKeyEncrypted(secretKey, password, Path.of
("aes_key_encrypted.txt"));

// Load encrypted
SecretKey loadedKey = KeyStorageUtil.loadAesKeyEncrypted(Path.of
("aes_key_encrypted.txt"), password);
```

Encrypted AES key storage uses AES-256 encryption with the provided password, making it safe for production use.

Key Format Conversion

Convert keys to/from PEM format

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair2048();

// Convert to PEM format
String pemPrivateKey = KeyStorageUtil.toPemFormat(keyPair.getPrivate().getEncoded(),
"PRIVATE KEY");
String pemPublicKey = KeyStorageUtil.toPemFormat(keyPair.getPublic().getEncoded(),
"PUBLIC KEY");

// Convert from PEM format
byte[] privateKeyBytes = KeyStorageUtil.fromPemFormat(pemPrivateKey);
byte[] publicKeyBytes = KeyStorageUtil.fromPemFormat(pemPublicKey);
```

Export and import public keys as Base64

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair2048();

// Export as Base64 string (convenient for sharing)
String base64PublicKey = KeyStorageUtil.exportPublicKeyAsBase64(keyPair.getPublic());

// Import from Base64 string
```

```
PublicKey importedKey = KeyStorageUtil.importPublicKeyFromBase64(base64PublicKey,
"RSA");
```

Base64 export is useful for sharing public keys via APIs, configuration files, or databases.

Key Validation

Validate RSA key strength

```
KeyPair keyPair = KeyGeneratorUtil.generateRsaKeyPair3072();

// Check if key meets minimum strength requirement
boolean isSufficient = KeyStorageUtil.isRsaKeySufficient(keyPair, 2048); // true

// Get key metadata
KeyStorageUtil.KeyMetadata metadata = KeyStorageUtil.getKeyMetadata(keyPair);
System.out.println("Algorithm: " + metadata.getAlgorithm()); // "RSA"
System.out.println("Key size: " + metadata.getKeySize());     // 3072
System.out.println("Format: " + metadata.getFormat());         // "PKCS#8"
```

`isRsaKeySufficient()` helps ensure your keys meet security requirements. NIST recommends minimum 2048-bit RSA keys, but 3072-bit or higher is preferred for long-term security.

Password Encoder

Key Derivation Functions (KDF)[12] from a password must be able to stand attacks like **brute-forcing**, **dictionary attacks**, **rainbow attacks** and more. Attempts to reverse hashed password values is common.

`ARGON2`, `BCRYPT`, `SCRYPT` and `PBKDF2` are common algorithms used for password hashing since they are much more robust when attacked.

`ARGON2id` (the hybrid variant combining Argon2i and Argon2d) is recommended as the most secure hash algorithm for passwords and is the default implementation for this API. It provides protection against both side-channel attacks and GPU/ASIC attacks.

Encoding

A plaintext password should always be encoded in case of a breach. After a password is encoded it may be stored for future matching.

Supported password encoders

```
    ARGON2("argon2"), // default
    BCRYPT("bcrypt"),
    SCRYPT("scrypt"),
    PBKDF2("pbkdf2");
```

Password encode using `CryptoUtil` *(default* `ARGON2`*)*

```
final String encodedPassword = CryptoUtil.encodePassword(password);
```

Alternative using `PasswordEncoderUtil` *directly*

```
// Default ARGON2
final String encodedPassword = PasswordEncoderUtil.encode(password);

// Or specify encoder type
final String encodedPassword = PasswordEncoderUtil.encode(password,
PasswordEncoderType.BCRYPT);
```

Matching

When a user is authenticated they must input their password. This plaintext password will be matched against the stored encoded password.

Password matching using `CryptoUtil` *(auto-detects encoder type)*

```
final boolean isMatching = CryptoUtil.matchesPassword(rawPassword, encodedPassword);
```

Alternative using `PasswordEncoderUtil` *directly*

```
// Auto-detects encoder type from {id} prefix
final boolean isMatching = PasswordEncoderUtil.matches(rawPassword, encodedPassword);

// Or specify encoder type explicitly for legacy systems
final boolean isMatching = PasswordEncoderUtil.matches(rawPassword, encodedPassword,
PasswordEncoderType.BCRYPT);
```

`matchesPassword()` automatically detects the encoder type from the `{id}` prefix (e.g., `{argon2}`, `{bcrypt}`), making it work seamlessly with passwords encoded using different algorithms.

Upgrading Passwords

When migrating from an older hash algorithm (like BCRYPT or SCRYPT) to a stronger one (like ARGON2), you can use the password upgrade feature. This allows gradual migration during user

login without requiring password resets.

Password upgrade using `CryptoUtil` *(upgrades to ARGON2)*

```
// During user login
String rawPassword = userInputPassword;
String storedEncodedPassword = getUserPasswordFromDatabase(); // e.g.,
"{bcrypt}$2a$10$..."

// First, verify the password
if (CryptoUtil.matchesPassword(rawPassword, storedEncodedPassword)) {
    // Login successful!

    // Check if password needs upgrade to ARGON2
    if (CryptoUtil.needsPasswordUpgrade(storedEncodedPassword)) {
        // Upgrade to ARGON2
        String upgradedPassword = CryptoUtil.upgradePassword(rawPassword,
storedEncodedPassword);

        // Save upgraded password back to database
        updateUserPasswordInDatabase(upgradedPassword);
    }
}
```

Alternative using `PasswordEncoderUtil` *directly*

```
// Check if password needs upgrade
if (PasswordEncoderUtil.needsUpgrade(storedEncodedPassword)) {
    // Upgrade to default (ARGON2)
    String upgradedPassword = PasswordEncoderUtil.upgradePassword(rawPassword,
storedEncodedPassword);

    // Or upgrade to specific type
    String upgradedPassword = PasswordEncoderUtil.upgradePassword(rawPassword,
storedEncodedPassword, PasswordEncoderType.ARGON2);
}
```

Password upgrades happen transparently during user login, ensuring a smooth migration path without disrupting users.

Documentation

PDF Version

A PDF version of this documentation is available:

- **Latest version:** Download PDF from GitHub Pages

- **Release versions:** Available as assets on the Releases page

Generate PDF locally

To generate a PDF version locally, you can use your IDE's AsciiDoc plugin or install `asciidoctor-pdf`:

```
# Install asciidoctor-pdf
gem install asciidoctor-pdf

# Generate PDF
asciidoctor-pdf README.adoc -o README.pdf
```

Recommended reading

- *Practical Cryptography for Developers*[17] by Svetlin Nakov
- *1Password - Why we moved to 256-bit AES keys*[18] by Jeffrey Goldberg in 2013

TODO

- Replacing MultipleMissingArgumentsError with own class without dependency to opentest4j and only throwing/supporting RuntimeException
- Support more algorithms for hashing digests[13] including password hashing
- Look into Diffie-Hellman and/or other better key exchange alternatives
- Consider adding support for generating and validation passwords with help of *Passay*[19] library

[1] https://en.wikipedia.org/wiki/Java_Cryptography_Architecture
[2] https://en.wikipedia.org/wiki/Java_Cryptography_Extension
[3] https://www.bouncycastle.org/
[4] https://en.wikipedia.org/wiki/RSA_(cryptosystem)
[5] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
[6] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[7] https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing
[8] https://github.com/codahale
[9] https://github.com/codahale/shamir
[10] https://en.wikipedia.org/wiki/Hash_function
[11] https://www.bouncycastle.org/java.html
[12] Key Derivation Functions. https://cryptobook.nakov.com/mac-and-key-derivation/kdf-deriving-key-from-password
[13] https://en.wikipedia.org/wiki/Secure_Hash_Algorithms
[14] https://crackstation.net/hashing-security.htm
[15] https://en.wikipedia.org/wiki/Digital_signature
[16] https://www.encryptionconsulting.com/overview-of-rsassa-pss
[17] https://cryptobook.nakov.com/
[18] https://blog.1password.com/why-we-moved-to-256-bit-aes-keys/
[19] http://www.passay.org