Crypto

Updated Nov 20, 2013 by mkobetic

This package provides cryptographic transform streams for hashing and encryption, secure random number generation and public key algorithm support. The algorithm implementations are reified into generic classes, e.g. Cipher or Hash, to allow supporting different implementation back-ends. Primary motivation is to allow exploiting available external cryptographic libraries. Currently we use the new bcrypt.dll on Windows (available on Vista and later) via the BCrypt class. On all other platforms we look for OpenSSL's libcrypto.so via the LibCryptoEVP class. See the implementation class comments for more details about required setup. Other implementations can be plugged in by following the example of the existing implementations.

Secure Random Number Generation

These capabilities are provided by the SecureRandomReadStream. The usage is very simple.

```
| random |
random := SecureRandom reading.
[       random read: 100
] ensure: [ random close ]
```

Hashing / Digests

The DigestRead/WriteStreams compute a cryptographic hash (MD5, SHA1, ...) of the bytes passing through the stream. The underlying streams must consume/produce bytes (0..255) and the digest streams themselves are binary as well. The digest value can be obtained using message #digest. It is possible to ask for the digest even before the stream is closed in which case the supporting Hash is cloned, the digest computation finished on the clone and the result returned, leaving the original Hash open to process more input. The size of the digest depends on the hash algorithm.

The set of supported algorithms depends on the underlying implementation. Each implementation class declares the algorithms it supports with class side methods with #algorithm: pragma. The pragma also defines the set of recognized names/aliases that can be used to refer to the algorithm. The set of supported algorithms will differ between different implementations although there should be significant overlap for the more common algorithms like MD5, SHA1, SHA256 and SHA512. Main difference between the algorithm is the size of the resulting digest. Larger digest generally means stronger (but often also slower) algorithm.

```
| sha |
sha := ByteArray new writing hashing: 'SHA256'.
(sha encoding: #ascii) write: 'Message in the bottle!'; close.
sha digest
```

Digest streams also support the keyed HMAC algorithm. It combines a hash algorithm with secret random key. It is used for data integrity protection, in some respects it is similar to a digital signature. The API is identical to regular digest streams, except the additional #key: parameter when the stream is created.

```
((ByteArray new: 10000 withAll: 42) reading
        hashing: 'SHA1'
        key: (ByteArray withAll: (1 to: 50))
) rest; close; digest
```

Encryption

The CipherRead/WriteStreams encrypt or decrypt bytes passing through the stream using pre-configured symmetric cipher (DES, AES, Blowfish, ...). The algorithm, direction (encrypt or decrypt), the key and initialization vector is configured when the stream is created. The underlying stream must be binary and the cipher streams themselves consume/produce bytes as well.

There are several required parameters for creation of cipher streams. The set of supported algorithms depends on the underlying implementation. Each implementation class declares the algorithms it supports with class side methods with #algorithm: pragma. Cipher modes are declared with class side methods with #mode: pragma. The pragmas also define the set of recognized names/aliases that can be used to refer to the modes or algorithms. The set of supported algorithms will differ between different implementations although there should be significant overlap for the more common algorithms like AES, DES, 3DES and RC4. Commonly available block cipher modes are CBC, CFB, OFB, CTR.

The second required parameter is the key, which is a ByteArray of appropriate size. Usually given cipher supports specific key size (e.g. DES - 8 bytes, 3DES - 24 bytes), or it can support multiple sizes (e.g. AES-128 - 16 bytes, AES-256 - 32 bytes). Finally, depending on cipher mode an initialization vector (IV) may be needed. It is again a ByteArray where the size corresponds to the block size of the cipher (e.g. DES - 8 bytes, 3DES - 8 bytes, AES - 16 bytes). Stream ciphers (e.g. RC4) don't need a mode or an IV. Following table lists some of the commonly used algorithms and their properties

algorithm	key size	block/iv size
AES	16B	16B
AES	24B	16B
AES	32B	16B
DES	8B	8B
3DES	24B	8B
RC4	8-256B	n/a

Here is an example using a stream cipher.

```
| random key encrypted decrypted |
random := SecureRandom reading.
[ key := random read: 16 ] ensure: [ random close ].
encrypted :=
	((ByteArray new writing encrypting: 'RC4' key: key) encoding: #ascii)
		write: 'Message in a bottle!'; close; terminal.
decrypted := (encrypted reading decrypting: 'RC4' key: key) encoding: #ascii.
[ decrypted rest ] ensure: [ decrypted close ]
```

In the case of block ciphers the content has to be processed in block-sized quantities, where block size depends on the cipher. This does not directly impact the behavior of the API but some amount of read-ahead from the source or write-behind to the destination is at times necessary. Similarly, write streams must be closed to make sure the last block was written. Note that overall data size must be a multiple of the block size of the cipher. Consequently it may have to be padded to make it so. Since the encrypted bytes do not necessarily carry the information about the actual amount of bytes written, it might be necessary to communicate the original size separately.

```
| random key iv message padding encrypted decrypted |
random := SecureRandom reading.
[	key := random read: 16.
	iv := random read: 16.
] ensure: [ random close ].
message := 'Message in a bottle!'.
padding := ByteArray new: (iv size - (message size \\ iv size)).
encrypted :=
	((ByteArray new writing encrypting: 'AES' mode: 'CBC' key: key iv: iv) encoding: #ascii)
		write: message; write: padding; close; terminal.
decrypted := (encrypted reading decrypting: 'AES' mode: 'CBC' key: key iv: iv) encoding: #ascii.
[ decrypted read: message size ] ensure: [ decrypted close ]
```

Public Key Algorithms

The public key cryptography API is not a streaming API but is a necessary complement to the cryptographic streams. It defines an abstract interface for public key algorithms. The key instances are either generated or imported. There are standard export formats for some kinds of keys but they aren't universally supported. This interface does not offer a comprehensive key import/export solution beyond a basic interface that allows creating keys from their constituent elements (usually a set of large positive integers). Other implementations may require their own specific ways of creating keys (e.g. a library for using keys embedded in smart cards). Primary focus of this API is using keys regardless of their underlying implementation. A key can be used for different purposes depending on the algorithm. An attempt to use a key for wrong purpose will most likely result in implementation specific error.

Available algorithms and capabilities depend on the underlying implementation. Implementation classes declare supported algorithms and capabilities with instance side methods with #algorithm:action: pragma. The method bodies define the associated action for given algorithm. Commonly supported algorithms are RSA, DSA and DH.

Signing (DSA, RSA)

To sign a sequence of bytes ("a message"), first a digest of the message has to be computed and then signed using the private key.

```
dsaPrivateKey := PrivateKey algorithm: 'DSA' size: 1024.
message := (ByteArray new writing encoding: #ascii) write: 'Message in a Bottle!'; close; terminal.
```

```
digest := (nil writing hashing: 'SHA1') write: message; close; digest.
signature := dsaPrivateKey sign: digest.
```

Specific signing algorithm may require specific type of digest, e.g. RSA can use MD5 or SHA1, DSA requires SHA1. Some algorithms support different ways to pad the input to the required size (generally determined by the size of the key). Unless there are no alternatives available for given algorithm message #sign:hash:padding: has to be used specifying which alternative should be used.

```
rsaPrivateKey := PrivateKey algorithm: 'RSA' size: 1024.
digest := (nil writing hashing: 'SHA256') write: message; close; digest.
signature := rsaPrivateKey sign: digest hash: 'SHA256' padding: 'PKCS1'
```

Corresponding public key has to be used to verify a signature. Again there are two variants of the verification method, with or without explicit hash and padding arguments subject to the same conditions as the signing ones. The result of the verification is a Boolean indicating if the signature checks out.

```
rsaPublicKey := rsaPrivateKey asPublicKey.
rsaPublicKey verify: signature of: digest hash: 'MD5' padding: 'PKCS1'
```

The object returned by the #sign: methods should be considered an opaque value with structure depending on algorithm and the underlying implementation. The structure does not matter if the signature will be used with same implementation. However, if the signature will be processed by different implementation or needs to be serialized for storage or transport the value will need to be converted into the constituent elements (usually large positive integers). Private keys support API to turn the signature value into its elements and public keys provide API to turn signature elements back into a signature value.

```
dsaPublicKey := dsaPrivateKey asPublicKey.
digest := (nil writing hashing: 'SHA1') write: message; close; digest.
signature := dsaPrivateKey sign: digest.
elements := dsaPrivateKey signatureElementsFrom: signature.
signature2 := dsaPublicKey signatureFromElements: elements.
"signature is equivalent to signature2"
```

Encryption (RSA)

With encryption capable algorithms (e.g. RSA) the public key can be used to encrypt a sequence of bytes which only the holder of the corresponding private key can decrypt back. Generally the size of the encrypted data cannot exceed the size of the keys. As with signatures, the bytes may have to be padded to key size and the padding method has to be specified.

```
encrypted := rsaPublicKey encrypt: message passing: 'PKCS1'.
decrypted := rsaPrivateKey decrypt: encrypted padding: 'PKCS1'.
"decrypted = message"
```

Key Agreement (DH)

Key agreement algorithms allow two parties to establish a shared secret over an unprotected channel. The two parties have to be in possession of compatible keys, where compatible means not just the same algorithm but also same (algorithm dependent) parameters. The two parties then exchange their public keys over the channel (they don't need to be encrypted, they are public) and then each use their private key and the other party's public key to arrive at the same, shared secret value. The secret value cannot be inferred by someone observing the public key exchange.

```
"Bob generates keys and sends public key to Alice"
bob := PrivateKey algorithm: 'DH' size: 1024.
bobPublic := bob asPublicKey.

"Alice generates keys using the same parameters, sends public key to Bob"
parameters := bobPublic elements first: 2. "p and g"
alice := PrivateKey algorithm: 'DH' parameters: parameters.
alicePublic := alice asPublicKey.

"Both parties derive the shared secret value using the other's public key"
bobSecret := bob derive: alicePublic.
aliceSecret := alice derive: bobPublic.
bobSecret = aliceSecret
```

Key Management

It is critical to properly release public and private keys after they are no longer needed. This is done by sending the #release message to the key. This action generally involves releasing any external resources but more importantly also properly discarding any associated secrets.

```
        dsaPrivateKey ifNotNil: [ :key | key release ].
        dsaPublicKey ifNotNil: [ :key | key release ].
        rsaPrivateKey ifNotNil: [ :key | key release ].
        rsaPublicKey ifNotNil: [ :key | key release ].
        alice ifNotNil: [ :key | key release ].
        alicePublic ifNotNil: [ :key | key release ].
        bob ifNotNil: [ :key | key release ].
        bobPublic ifNotNil: [ :key | key release ].
```

Enter a comment: Hint: You can use Wiki Syntax.

Submit