Crittografia in C#

Introduzione alle classi hashing, crittografia simmetrica e asimmetrica

.NET 4.0/C# 4.0

Indice generale

1 Introduzione alle funzioni di crittografia

Questo tutorial introduce le classi che implementano algoritmi di ***hashing*** (o ***message digest***), **crittografia simmetrica**; e **crittografia asimmetrica**. Tute le classi sono definite nel *namespace* `System.Security.Cryptografy`.

1.1 Algoritmi di hashing

I più famosi sono MD5 e SHA e sono impiegati per produrre una "impronta" (***digest*** *o* ***hash***) di lunghezza fissa a partire da un messaggio di lunghezza variabile. Sono applicati per:

- Verificare l'integrità di un messaggio, confrontando il suo *digest* prima e dopo la trasmissione.
- Verificare un cambiamento del contenuto di un file.
- Cifrare le password.
- Generare chiavi/cifrare dati nell'ambito di protocolli di autenticazione e/o comunicazione protetta.

1.2 Crittografia simmetrica

Consente di cifrare un messaggio e successivamente decifrarlo *utilizzando la stessa chiave*. È utilizzata per:

- Cifrare documenti mediante password.
- Cifrare password quando è richiesta la possibilità di recuperare il valore originale,
- Proteggere la comunicazione in rete.

Gli algoritmi simmetrici richiedono che la chiave sia conosciuta soltanto dall'utente e/o dall'applicazione che esegue i processi di cifratura e decifratura. Questo è un problema soprattutto nella comunicazione di rete, poiché è necessario che i soggetti coinvolti condividano la stessa chiave.

1.3 Crittografia asimmetrica

È conosciuta anche come **crittografia a chiave pubblica** e utilizza una coppia di chiavi, una **privata** (segreta) e l'altra **pubblica**. Questo tipo di crittografia fornisce la soluzione ottimale a due problemi specifici:

- Scambio di chiavi in un canale non sicuro.
- **Firma digitale**: verifica dell'autenticità di un mittente.

1.3.1 Scambio di chiavi

Rappresenta un problema fondamentale della comunicazione su un canale non sicuro. In genere, due soggetti comunicano in modo sicuro cifrando i messaggi mediante un algoritmo simmetrico; ma ciò presuppone che usino entrambi la stessa chiave, che, in molti scenari, deve essere comunicata prima che sia attuata la cifratura. Gli algoritmi asimmetrici forniscono due soluzioni:

- Uno dei due soggetti genera la chiave simmetrica e usa la chiave pubblica dell'altro per cifrarla prima di inviargliela. L'altro ottiene quindi in modo sicuro la chiave simmetrica da usare per avviare la comunicazione vera e propria.
- Mediante l'algoritmo **Diffie-Hellman** i due soggetti si scambiano le rispettive chiavi pubbliche e con esse generano in proprio la chiave simmetrica, che sarà la stessa per entrambi.

1.3.2 Firma digitale (signature)

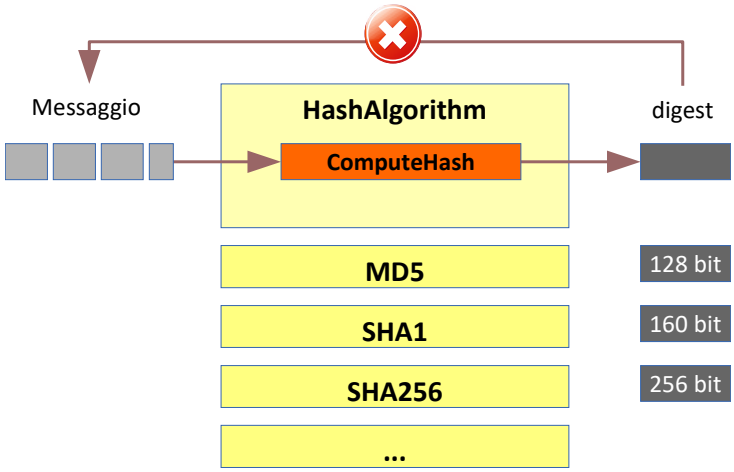
Gli algoritmi asimmetrici consentono di autenticare un messaggio accodando ad esso un *digest* cifrato con la chiave privata del mittente. Il ricevente decifrerà il *digest* utilizzando la chiave pubblica del mittente, ma l'operazione avrà successo soltanto se il mittente è veramente il proprietario della chiave pubblica. Naturalmente, l'intero processo si basa sulla premessa che la chiave pubblica sia certificata da un ente apposito che ne garantisca la validità.

2 Classi di hashing

Gli algoritmi di *hashing* condividono le seguenti caratteristiche generali:

- Sono **non reversibili**: non è possibile ottenere il messaggio originale a partire dal suo *digest* (***one-way***).
- Il *digest* ha dimensione fissa, indipendentemente dalla lunghezza del messaggio da cifrare.
- Messaggi quasi uguali producono *digest* molto diversi.
- Due messaggi diversi *non possono* produrre lo stesso *digest*[1].

La maggior parte di questi algoritmi non utilizza una chiave di cifratura, dunque produce sempre lo stesso *digest* a partire dallo stesso messaggio.



2.1 Class SHA1Managed

.NET definisce svariate classi che implementano algoritmi di *hashing* e derivano tutte dalla classe astratta `HashAlgorithm`. Qui considero la classe `SHA1Managed`, il cui *digest* è di 20 byte. La classe definisce un metodo, `ComputeHash()`, che riceve il messaggio da cifrare e produce il *digest* corrispondente. Sia messaggio che *digest* sono rappresentati mediante un vettore di byte.

```
string msg = "testo da cifrare!";
byte[] msgData = Encoding.UTF8.GetBytes(msg);

SHA1Managed sha = new SHA1Managed();
byte[] digest = sha.ComputeHash(msgData);
//-> A4-F6-E5-2F-BB-76-2D-D0-4A-AC-9C-2F-65-D4-FB-F0-15-75-56-73
```

2.1.1 Rappresentare il digest in forma di stringa: base64string

In alcuni casi è necessario memorizzare il *digest* in formato stringa, ad esempio per memorizzarlo in un file di testo, un campo di un database, o inviarlo su una connessione di rete. In questo caso è importante convertirlo in ***base64string***. Questo formato utilizza un sotto insieme del codice ASCII per codificare una sequenza di byte in stringa.

1 In realtà possono verificarsi delle collisioni: due messaggi distinti producono lo stesso *digest*.

Per convertirlo si usa il metodo statico `ToBase64String()` della classe `Convert`:

```
...
byte[] digest = sha.ComputeHash(msgData);
string digest64 = Convert.ToBase64String(digest);
//-> = pPblL7t2LdBKrJwvZdT78BV1VnM=
```

Nota bene: per convertire il *digest* si potrebbe usare `Encoding.UTF8.GetString()`; ebbene, non è una scelta corretta; infatti, il *digest* potrebbe contenere dei valori che non sono codificabili in UTF8. Inoltre, il risultato potrebbe contenere i caratteri `\r` e `\n`, i quali sarebbero interpretati come terminatori di riga nel caso lettura mediante il metodo `ReadLine()`.

Il formato *base64string* non usa questi caratteri e dunque non presenta questo problema.

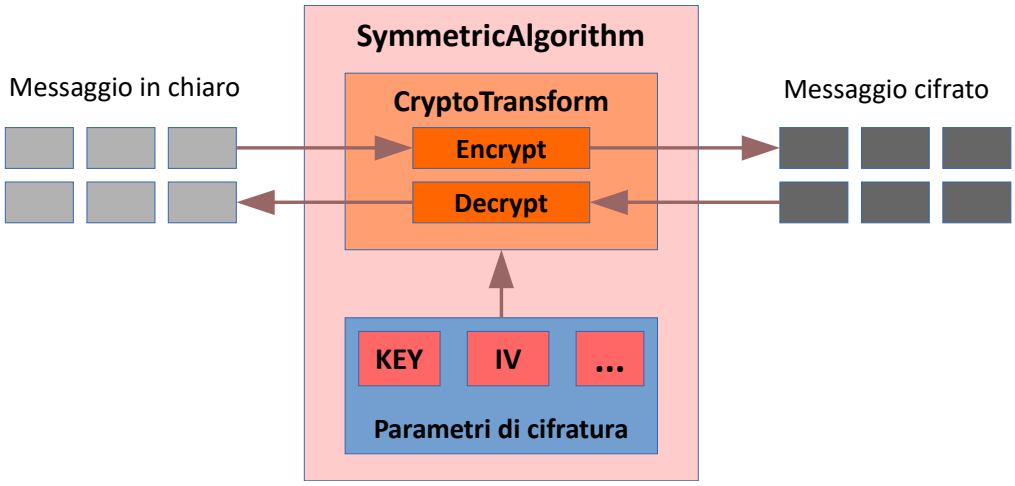
3 Classi di crittografia simmetrica

Gli algoritmi di crittografia simmetrica condividono le seguenti caratteristiche:

- Sono **reversibili**: è possibile ottenere il messaggio originale a partire da quello cifrato (***two-way***).
- Utilizzano la stessa chiave in entrambe le operazioni, cifratura e decifratura.
- La chiave ha un dimensione fissa e, di norma, determina la sicurezza dell'algoritmo.

3.1 Caratteristiche di un oggetto di crittografia simmetrica

Tutte le classi di crittografia simmetrica derivano dalla classe astratta `SymmetricAlgorithm` e dunque condividono la stessa struttura:



3.1.1 Parametri di cifratura

Influenzano il risultato delle operazioni di cifratura e decifratura. I più importanti sono la **chiave** e il **vettore di inizializzazione**, entrambi memorizzati come vettori di byte. Hanno una dimensione fissa che dipende dal tipo di algoritmo:

Algoritmo	Lunghezza predefinita (bit)		Lunghezze ammesse (bit)	
	Chiave	Blocco	Chiave	Blocco
DES	64	64	64	64
RC2	128	64	40 ↔ 128	64
TripleDES	192	64	128 ↔ 192	64
AES	256	128	128 ↔ 256	128
Rijndael	256	128	128 ↔ 256	128 ↔ 256

Un maggior numero di bit implica una maggior sicurezza al costo di una minor efficienza.

3.1.2 CryptoTransform

L'oggetto ***CryptoTransform*** esegue il processo di cifratura. Esiste in due "declinazioni", una per cifrare, l'altra per decifrare. Può essere usato direttamente per cifrare un blocco di byte, oppure in congiunzione con un **CryptoStream**.

3.2 Classe DESCryptoServiceProvider

L'uso di un oggetto di crittografia simmetrica vede due fasi: l'impostazione dei parametri di cifratura e il processo di cifratura vero e proprio.

3.2.1 Impostare i parametri di cifratura

Il codice seguente imposta chiave e vettore di inizializzazione:

```
DESCryptoServiceProvider des = new DESCryptoServiceProvider();
byte[] key = new byte[8] { 49, 50, 51, 52, 53, 54, 55, 56 }; // "12345678"
des.Key = key;
des.IV = key;  // imposta il vettore uguale alla chiave (non obbligatorio, ma possibile)
```

In molti scenari è l'utente a stabilire la chiave, ad esempio mediante una password:

```
string pw;
//... l'utente inserisce la password, che viene usata come chiave
byte[] key = Encoding.ASCII.GetBytes(pw);
des.Key = key;   <--potrebbe sollevare un errore!
des.IV = key;
```

Nota bene: il codice evidenziato solleva un problema: non esiste garanzia che la chiave così ottenuta sia della giusta dimensione. Vedremo più avanti come gestire questo problema.

3.2.2 Processo di cifratura

Occorre innanzitutto ottenere un oggetto di tipo `ICryptoTransform`, capace di cifrare o decifrare una sequenza di byte. Quest'ultimo definisce il metodo `TransformFinalBlock()`, che richiede il vettore da cifrare:

```
...
string plainText = "testo in chiaro";
byte[] plainData = Encoding.ASCII.GetBytes(plainText);
ICryptoTransform enc = des.CreateEncryptor();
byte[] encData = enc.TransformFinalBlock(plainData, 0, plainData.Length);

//chiaro    -> 74-65-73-74-6F-20-69-6E-20-63-68-69-61-72-6F
//cifrato   -> 3D-5E-71-0C-B0-AA-60-26-2E-1C-C8-32-89-C8-B0-5A
```

Nota bene: l'oggetto usato per cifrare è un *encryptor* e viene ottenuto mediante il metodo `CreateEncryptor()`.

3.2.3 Processo di decifratura

È del tutto simile al precedente, con la differenza che occorre utilizzare un *decryptor*:

```
string plainText = "testo in chiaro";
// ... qui cifra il testo (vedi esempio precedente)
ICryptoTransform dec = des.CreateDecryptor();
byte[] decData = dec.TransformFinalBlock(encData, 0, encData.Length);
string decText = Encoding.ASCII.GetString(decData); //-> "testo in chiaro"
```

3.2.4 Usare la codice Base64string

Se si rende necessario memorizzare i dati cifrati in forma di stringa, è necessario usare il formato **base64string** (2.1.1); infatti, non sempre gli *encoder* UTF8 e ASCII codificano correttamente i dati cifrati. Applicando questi ultimi è possibile che i processi di cifratura e decifratura non lavorino in modo simmetrico e producano risultati incompatibili.

Segue lo schema generale da adottare nel caso in cui si scelga di memorizzare i dati cifrati in formato testo.

```
string Cifra(string plainText)
{
    byte[] plainData = Encoding.ASCII.GetBytes(plainText); // anche UTF8 va bene
    //... CIFRA plainData -> byte[] encData
    return Convert.ToBase64String(encData);
}

string Decifra(string encText)
{
    byte[] encData = Convert.FromBase64String(encText);
    //... DECIFRA encData -> byte[] plainData
    return Encoding.UTF8.GetBytes(plainData);
}
```

Nota bene: dal punto di vista della codifica `string` → `byte[]` e viceversa, i due processi non sono perfettamente identici.

3.2.5 Impostare la dimensione della chiave

Se la chiave è una password, prima di usarla per impostare la proprietà `Key` è necessario codificarla in un vettore di byte della lunghezza corretta. Due soluzioni sono:

- Codificare la stringa in un vettore di byte e quindi adattarne "manualmente" la dimensione, troncandolo o aggiungendo byte di riempimento.
- Codificare la stringa in un vettore di byte e quindi ottenere un *digest* utilizzando un algoritmo di *hashing*, infine adattare la lunghezza se necessario.

La seconda soluzione è senz'altro molto più sicura:

```
string pw = "il mio gatto si chiama Cippa Lippa";  // password inserita dall'utente
byte[] pwData = Encoding.ASCII.GetBytes(pw);

// crea un digest della password
```

```
SHA1Managed sha = new SHA1Managed();
byte[] pwHash= sha.ComputeHash(pwData, 0, pwData.Length); // 20 byte

DESCryptoServiceProvider des = new DESCryptoServiceProvider();
// tronca il digest alla dimensione richiesta (in questo caso 8 byte)
Array.Resize(ref pwHash, des.KeySize / 8);//    64 / 8 --> 8 byte
des.Key = pwHash;
des.IV = pwHash;
...
```

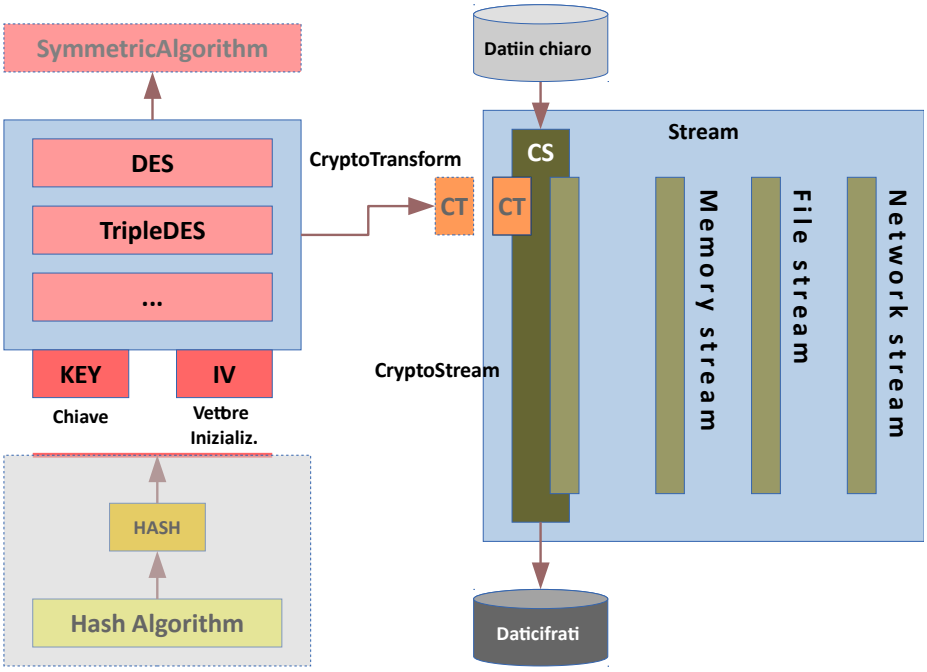
Ci sono alcune considerazioni da fare:

- Per l'algoritmo scelto (DES) `Key` e `IV` hanno uguale lunghezza, ma potrebbe non essere così. In questo caso occorre creare due vettori della lunghezza desiderata.
- Conviene scegliere un algoritmo di *hash* in base al tipo di crittografia da usare. Ad esempio, se si intende cifrare con **Rijndael** conviene scegliere **SHA256**, poiché produce un *digest* della lunghezza giusta per la chiave.

4 Cifrare mediante uno stream: CryptoStream

In molti scenari l'uso diretto di un **CryptoTransform** non è l'ideale, poiché i dati da cifrare possono essere gestiti in modo più naturale mediante uno stream. Ciò accade, per esempio, quando si deve cifrare un file di testo, un file binario, oppure un flusso di dati proveniente da una rete. A questo scopo esiste la classe `CryptoStream`.

Un `CryptoStream` è in grado di scrivere i dati su uno stream utilizzando un `CryptoTransform` per cifrarli. Discorso analogo vale per la lettura. Lo schema sottostante riassume la modalità d'uso di un `CryptoStream`.



Un `CryptoStream` "incorpora" un oggetto di cifratura (CT, nello schema) e uno stream nel quale scriverà (o leggerà) i dati. Una volta creato il `CryptoStream` sarà possibile utilizzarlo come qualsiasi altro stream, ad esempio scrivendo o leggendo su di esso mediante uno `StreamWriter` o uno `StreamReader`.

4.1 Uso della classe CryptoStream: cifrare un file di testo

Problema: si desidera cifrare un file di testo mediante una password.

Una soluzione è la seguente:

Leggi il file in memoria (File.ReadAllLines())
Crea e imposta un oggetto di crittografia
Ottieni un CryptoTransform (Encryptor)
Crea un FileStream che punti al file che memorizzerà il testo cifrato (file da scrivere)
Crea un CryptoStream che usi l'Encryptor e il FileStream
Crea un StreamWriter per scrivere sul CryptoStream
Usa lo StreamWriter, scrivendo le righe precedentemente caricate in memoria.

Segue il codice (l'argomento `outputPath` punta al file da creare):

```
public static void EncryptTextFile(string pw, string filePath, string outputPath)
{
    string[] textLines = File.ReadAllLines(filePath);

    // CREA E IMPOSTA L'OGGETTO DI CRITTOGRAFIA SIMMETRICA
    DESCryptoServiceProvider des = new DESCryptoServiceProvider();

    // Imposta la chiave mediante digest (vedi paragrafo 3.2.4)
    byte[] pwHash = CreatePasswordDigest(pw, des.KeySize);
    des.Key = pwHash;
    des.IV = pwHash;

    // CREA IL CRYPTOTRANFORM, IL FILESTREAM, IL CRYPTOSTREAM E LO STREAMWRITER
    ICryptoTransform ct = des.CreateEncryptor();
    FileStream fs = new FileStream(outputPath, FileMode.Create);
    CryptoStream cs = new CryptoStream(fs, ct, CryptoStreamMode.Write);

    // SCRIVE IL TESTO NEL NUOVO FILE
    StreamWriter sw = new StreamWriter(cs);
    foreach (var line in textLines)
    {
        sw.WriteLine(line);
    }
    sw.Close();
}
```

Il codice evidenziato mostra le relazioni che intercorrono tra gli oggetti.

Lo `StreamWriter` scrive nel `CryptoStream` `cs`; questo scrive nel `FileStream` `ft` utilizzando il `CryptoTransform` `ct` per cifrare.

4.1.1 Leggere il file cifrato

Il procedimento per decifrare il file è del tutto analogo, con la differenza che si usa un *decryptor* e uno `StreamReader`.

```
public static IEnumerable<string> DecryptTextFile(string pw, string filePath)
{
    // CREA E IMPOSTA L'OGGETTO DI CRITTOGRAFIA SIMMETRICA
    DESCryptoServiceProvider des = new DESCryptoServiceProvider();
    byte[] pwHash = CreatePasswordDigest(pw, des.KeySize);
    des.Key = pwHash;
    des.IV = pwHash;

    // CREA IL CRYPTOTRANFORM, IL FILESTREAM, IL CRYPTOSTREAM E LO STREAMREADER
    ICryptoTransform ct = des.CreateDecryptor();
    FileStream fs = new FileStream(filePath, FileMode.Open);
    CryptoStream cs = new CryptoStream(fs, ct, CryptoStreamMode.Read);

    // LEGGE IL TESTO DAL FILE
    StreamReader sr = new StreamReader(cs);
```

```
    string line = sr.ReadLine();
    while (line != null)
    {
        yield return line;
        line = sr.ReadLine();
    }
    sr.Close();
}
```

Nel codice sono evidenziate le piccole differenze di impostazione nella creazione del `FileStream` e del `CryptoStream`.

5 Classi di crittografia asimmetrica

.NET prevede diverse classi di crittografia asimmetrica e derivano tutte dalla classe astratta `AsymmetricAlgorithm`:

- `ECDsaCng`: usata principalmente per la firma digitale;
- `ECDiffieHellmanCng`: usata principalmente per lo scambio di chiavi;
- `RSACryptoServiceProvider`: fornisce servizi generali di crittografia e dunque include le funzioni precedenti.

Pur condividendo delle caratteristiche in comune, le tre classi offrono servizi diversi e, di norma, non sono utilizzabili intercambiabilmente.

5.1 Scambio di chiavi simmetriche: ECDiffieHellmanCng

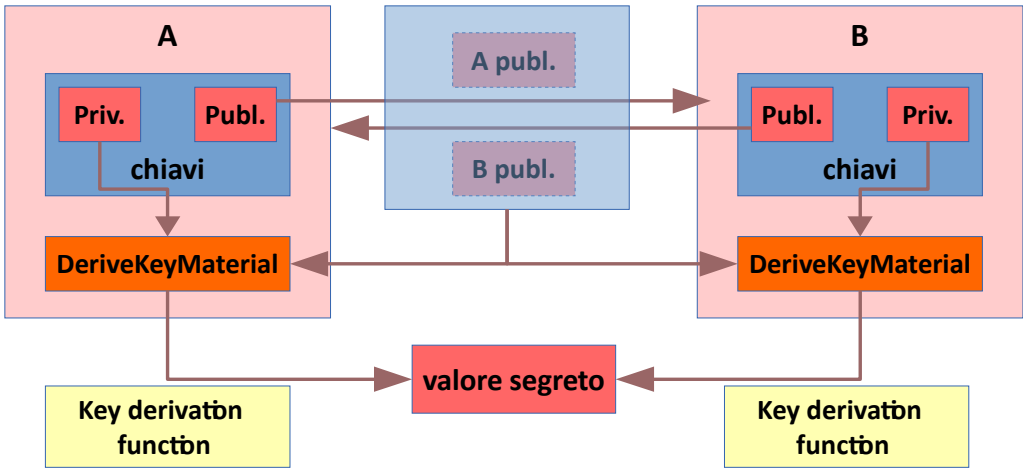
Questa classe implementa un processo che consente a due soggetti di scambiarsi un valore segreto (***secret agreement***) attraverso un canale non protetto. In realtà il valore non viaggia nel canale, ma viene calcolato da entrambi i soggetti.

La classe consente di stabilire dei parametri di *post-processing* (***key derivation function***) del valore segreto, in modo che questo rispetti dei requisiti (sia cifrato, abbia una certa lunghezza, etc).

5.1.1 Meccanismo di generazione del secret agreement

Ipotizziamo l'esistenza di due oggetti di tipo `ECDiffieHellmanCng`, **A** e **B**. Entrambi hanno già la propria coppia di chiavi (privata e pubblica). Per generare il valore segreto devono semplicemente scambiarsi le chiavi pubbliche e accordarsi sul tipo di post-processing da applicare.

La chiamata al metodo `DeriveKeyMaterial()` produce il valore segreto sotto forma di un vettore di byte. I vettori prodotti sono identici.



Il codice che segue mostra questo meccanismo in azione:

```
ECDiffieHellmanCng A = new ECDiffieHellmanCng();  // coppia di chiavi autogenerata
ECDiffieHellmanCng B = new ECDiffieHellmanCng();  // coppia di chiavi autogenerata
```

```
// Key derivation function predefinita: HASH - SHA256 -> valore segreto di 32 byte

byte[] AsecretKey= A.DeriveKeyMaterial(B.PublicKey);
byte[] BsecretKey = B.DeriveKeyMaterial(A.PublicKey);

// AsecretKey -> 03-F9-C5-8A-E9-B6-59-0E-D6-23-F7-E5-4D-C9-5C-08-06-B2-C5-8C...
// BsecretKey -> 03-F9-C5-8A-E9-B6-59-0E-D6-23-F7-E5-4D-C9-5C-08-06-B2-C5-8C...
```

Nota bene: nella chiamata al metodo `DeriveKeyMaterial()` ogni oggetto utilizza la chiave pubblica dell'altro.

5.1.2 Scambio chiavi come vettore di byte

Il precedente codice presuppone che ogni oggetto abbia accesso alla chiave pubblica dell'altro; si tratta di un presupposto poco realistico. Negli scenari reali, due soggetti devono comunicare attraverso una connessione di rete e dunque inviarsi le chiavi come flussi di byte. Dunque:

- È necessario trasformare la chiave in un vettore di byte e inviarlo attraverso la rete;
- È necessario riprodurre la chiave dell'altro a partire dal vettore di byte ricevuto.

Il codice seguente simula questo scenario[2]:

```
ECDiffieHellmanCng A = new ECDiffieHellmanCng();
byte[] AbyteKey = A.PublicKey.ToByteArray();
//... spedisce il vettore bella rete

ECDiffieHellmanCng B = new ECDiffieHellmanCng();
byte[] BbyteKey = B.PublicKey.ToByteArray();
//... spedisce il vettore bella rete

//...legge il vettore dalla rete e crea una chiave di tipo CngKey (sia A che B)
CngKey Akey = CngKey.Import(AbyteKey, CngKeyBlobFormat.EccPublicBlob);
CngKey Bkey = CngKey.Import(BbyteKey, CngKeyBlobFormat.EccPublicBlob);

//crea i valori segreti (sia A che B)
byte[] BsecretKey = B.DeriveKeyMaterial(Akey);
byte[] AsecretKey = A.DeriveKeyMaterial(Bkey);

// AsecretKey -> 4E-FD-D0-EC-6B-57-08-12-70-2F-6F-61-53-04-94-F4-9E-BB-83-D2...
// BsecretKey -> 4E-FD-D0-EC-6B-57-08-12-70-2F-6F-61-53-04-94-F4-9E-BB-83-D2...
```

Nota bene: il metodo statico `Import()` della classe `CngKey` crea una chiave a partire da un vettore di byte. Questa viene passata al metodo `DeriveKeyMaterial()` per produrre il valore segreto.

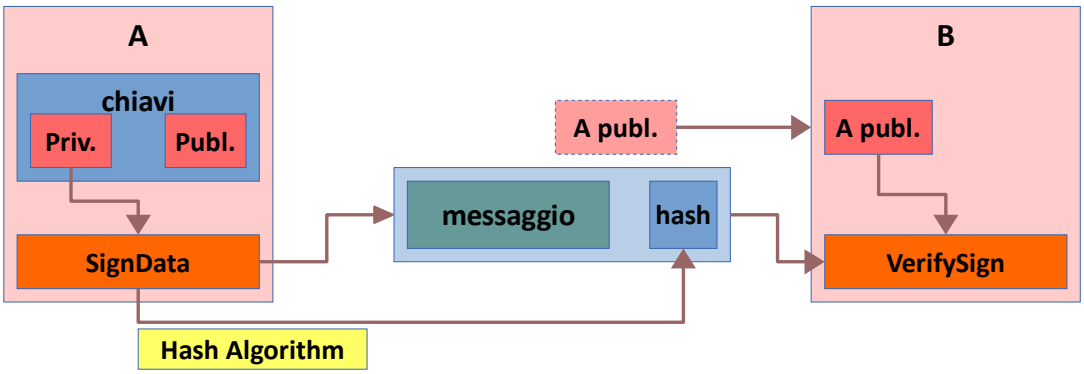
5.2 Firma digitale: classe RSACryptoServiceProvider

La classe `RSACryptoServiceProvider` definisce le funzionalità necessarie per firmare un messaggio e verificare l'autenticità della forma. Queste sono definite dai metodi `SignData()` e `VerifyData()`: il primo viene usato dal mittente per firmare, il secondo dal ricevente per verificare l'autenticità della firma.

2 Esiste un'alternativa che non usa il tipo **CngKey**, ma è praticamente identica.

5.2.1 Meccanismo di firma e verifica autenticità

Immaginiamo che **A** debba inviare un messaggio a **B** garantendo l'autenticità del mittente. Per farlo, A crea un *digest* del messaggio e poi cifra quest'ultimo con la propria chiave privata. Dopodiché spedisce messaggio+*digest* a B. Quest'ultimo utilizza la chiave pubblica di A per decifrare il *digest*, quindi produce a sua volta un *digest* del messaggio e lo confronta con quello ricevuto, verificandone l'uguaglianza.



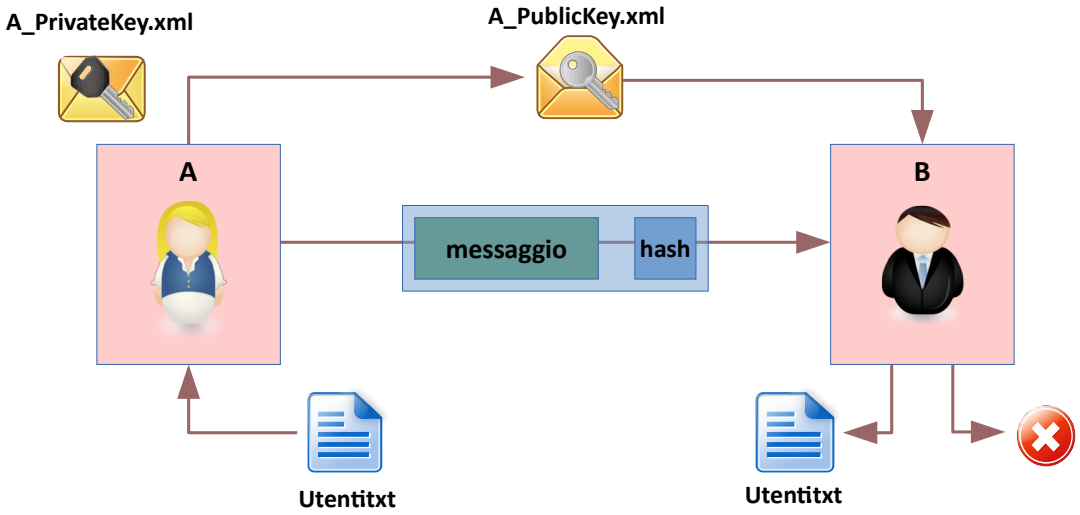
Naturalmente, A e B devono utilizzare lo stesso algoritmo di hashing.

> **Cifratura del messagio come prova di autenticità**
>
> L'uso di un *digest* non sarebbe essenziale, poiché A e B potrebbero cifrare e decifrare direttamente il messaggio. D'altra parte, la crittografia asimmetrica è meno efficiente dell'*hashing*; dunque viene creato un *digest* del messaggio e cifrato quest'ultimo invece di elaborare direttamente il messaggio.
>
> La classe `RSACryptoServiceProvider` crea un *digest* di 128 byte.

5.3 Esempio di firma e successiva autenticazione

Si vuole simulare l'invio di un documento da **A** a **B** in modo che quest'ultimo possa autenticare l'identità del mittente. Il documento non necessita di essere cifrato. La chiave pubblica di A è salvata in un file liberamente accessibile a B. La chiave privata di A è salvata su un secondo file, accessibile soltanto ad A

Lo schema sottostante riassume lo scenario.



5.3.1 Pseudo codice del procedimento di firma e autenticazione

Dividiamo il procedimento in tre parti.

- Generazione delle chiavi pubblica e privata di A. (In uno scenario realistico entrambe esistono già e la seconda è accessibile presso un provider pubblico.)
- Firma del documento.
- Verifica del documento firmato.

- Generazione delle chiavi

Crea RSACryptoServiceProvider

Genera chiave pubblica e privata e salvale su disco

- Firma (eseguita da A)

*Crea **RSACryptoServiceProvider***

Importa la chiave privata precedentemente generata

*Carica documento e codificalo in un vettore di byte → **messaggio***

*Firma il documento (**SignData()**) → vettore di byte contenente la **firma***

Scrivi (firma + documento) su stream

> *In uno scenario reale sarebbe usato un NetworkStream. Qui possiamo usare un MemoryStream e ritornare come risultato un vettore di byte.*

- Verifica documento firmato (riceve messaggio sotto forma di vettore di byte)

*Crea **RSACryptoServiceProvider***

Importa la chiave pubblica precedentemente generata

Crea un MemoryStream contenente il vettore di byte del messaggio

In uno scenario reale si leggerebbe da un NetworkStream.

Leggi sia firma che messaggio da stream

*Verifica autenticità firma (**VerifySign()**)-→ bool*

5.3.2 Implementazione del procedimento

Segue il codice generale:

```
string A_PubPath = @"Keys\A_PublicKey.xml";
string A_PrivPath = @"Keys\A_PrivateKey.xml";

GeneratePrivatePublicKey(A_PubPath, A_PrivPath);

byte[] messageSigned = Sign(@"Data\utenti.txt", A_PrivPath);

bool ok = VerifySign(A_PubPath, messageSigned);
```

Segue il codice che genera e salva le chiavi:

```
public static void GeneratePrivatePublicKey(string pubKeyFile, string privKeyFile)
{
    var A = new RSACryptoServiceProvider();
    string xmlPubKey = A.ToXmlString(false);      // genera chiave pubblica
    string xmlPrivKey = A.ToXmlString(true);      // genera chiave privata
    File.WriteAllText(pubKeyFile, xmlPubKey);
    File.WriteAllText(privKeyFile, xmlPrivKey);
}
```

Nota bene: le chiavi vengono generate in formato XML e salvate separatamente su due file. Questa è soltanto una possibilità: la classe `RSACryptoServiceProvider` fornisce altri meccanismi per esportare e importare le chiavi.

Segue il codice che firma il documento:

```
public static byte[] Sign(string filePath, string privKeyFilePath)
{
    var A = new RSACryptoServiceProvider();
    string privKey = File.ReadAllText(privKeyFilePath);
    A.FromXmlString(privKey);       // importa la chiave privata

    string message = File.ReadAllText(filePath);
    byte[] messageBytes = Encoding.UTF8.GetBytes(message);
    // firma il messaggio; signBytes contiene un digest cifrato con la chiave privata
    byte[] signBytes = A.SignData(messageBytes, new SHA1Managed());

    // crea stream e scrive FIRMA + MESSAGGIO (in quest'ordine)
    MemoryStream ms = new MemoryStream();
    BinaryWriter bw = new BinaryWriter(ms);
    bw.Write((byte)signBytes.Length);      //usa un solo byte per la lunghezza
    bw.Write(signBytes);
    bw.Write(message.Length);
    bw.Write(messageBytes);
    bw.Close();
```

```
    return ms.ToArray();
}
```

Infine il codice di ricezione del messaggio e di verifica della firma. Questo presuppone che il vettore data contenga già il messaggio spedito dal mittente:

```
public static bool VerifySign(string pubKeyFilePath, byte[] data)
{
    MemoryStream ms = new MemoryStream(data);

    var B = new RSACryptoServiceProvider();
    string pubKey = File.ReadAllText(pubKeyFilePath);
    B.FromXmlString(pubKey);// importa chiave pubblica

    BinaryReader br = new BinaryReader(ms);
    int signLenght = br.ReadByte();
    byte[] signature = br.ReadBytes(signLenght);
    int messageLength = br.ReadInt32();
    byte[] messageBytes = br.ReadBytes(messageLength);

    // verifica autenticità del mittente
    bool ok = B.VerifyData(messageBytes, new SHA1Managed(), signature);
    return ok;
}
```