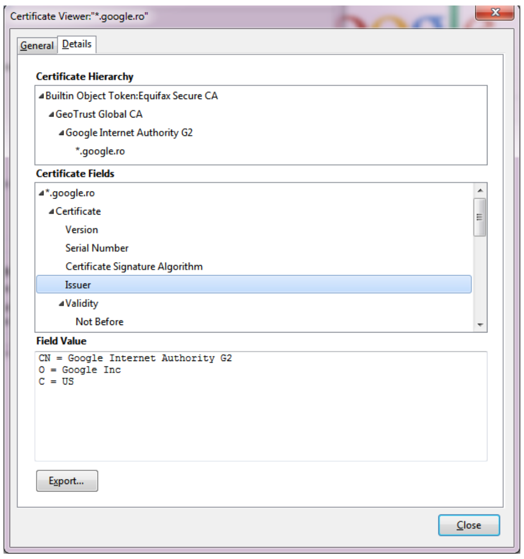
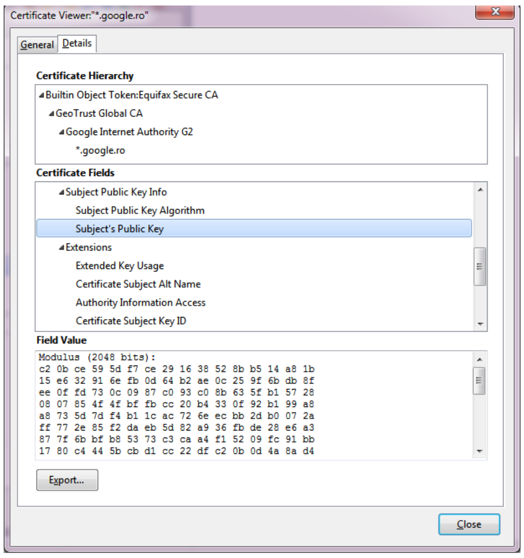
Chapter 4. THE RSA PUBLIC-KEY CRYPTOSYSTEM IN .NET

This section presents the RSA cryptosystem based on its embodiment from the .NET framework. The name RSA stems from the name of the three inventors: Rivest, Shamir and Adleman, who published the cryptosystem in 1978. RSA can be used to perform public key encryptions as well as digital signatures. The applicative target is quite distinct for the two operations: public key encryptions are generally used to encrypt keys for symmetric cryptosystem (you can use public keys to encrypt messages or files, but this would be highly inefficient) while digital signatures are used to prove that a piece of data originates from a particular entity. For example, you use Google's public certificate to retrieve its public key and then encrypt a smaller AES key with the public key in order to create an encrypted tunnel between your e-mail client and Gmail's server. The reason for encrypting a small session key with the RSA, rather than encrypting messages that are exchanged between parties is simple: efficiency. Indeed, RSA has the benefit of not requiring a secret key shared between parties, but it is in several orders of magnitude less efficient than symmetric algorithms such as AES. For this reason, RSA encryption is generally used for exchanging small secret keys for AES, 3DES, etc. To develop our example further, the piece of data from Google that you used as a public key needs to be signed by a trusted party, recognized by your browser, in order to make sure that indeed it belongs to Google (otherwise a man-in-the-middle attack can be mounted). Figure 1 shows parts of such a certificate.



```
CN = Google Internet Authority G2
O = Google Inc
C = US
```



Certificate Viewer:"*.google.ro"
General
Details
Certificate Hierarchy
Builtin Object Token:Equifax Secure CA
GeoTrust Global CA
Google Internet Authority G2
*.google.ro
Certificate Fields
Subject Public Key Info
Subject Public Key Algorithm
Subject's Public Key
Extensions
Extended Key Usage
Certificate Subject Alt Name
Authority Information Access
Certificate Subject Key ID
Field Value
Export...
Close

```
Modulus (2048 bits):
c2 0b ce 59 5d f7 ce 29 16 38 52 8b b5 14 a8 1b
15 e6 32 91 6e fb 0d 64 b2 ae 0c 25 9f 6b db 8f
ee 0f fd 73 0c 09 87 c0 93 c0 8b 63 5f b1 57 28
08 07 85 4f 4f bf fb cc 20 b4 33 0f 92 b1 99 a8
a8 73 5d 7d f4 b1 1c ac 72 6e ec bb 2d b0 07 2a
ff 77 2e 85 f2 da eb 5d 82 a9 36 fb de 28 e6 a3
87 7f 6b bf b8 53 73 c3 ca a4 f1 52 09 fc 91 bb
17 80 c4 44 5b cb d1 cc 22 df c2 0b 0d 4a 8a d4
```

Figure 1. Portion of an RSA certificate issued for Google, note issuer on the left and part of the public key on the right

4.1 BRIEF THEORETICAL BACKGROUND

You are referred to the lecture material for more details on the RSA. However, to make things clearer, we make a brief recap on how RSA works.

How text-book RSA encryption works. As any public key cryptosystem, the RSA is a collection of three algorithms:

- ***Key generation:*** generate two random primes p, q then compute: $n = pq, \varphi(n) = (p-1)(q-1)$ fix a public exponent e such that $\gcd(e, \varphi(n)) = 1$ then compute $d = e^{-1} \bmod \varphi(n)$.
- ***Encrypt:*** given the message m and the public key $Pb = (n, e)$, encrypt the message as $c = m^e \bmod n$.
- ***Decypt:*** given the ciphertext c and the private key $Pv = (n, d)$, decrypt the message as $m = c^d \bmod n$.

How text-book RSA signature works. The key generation procedure is identical to the RSA encryption scheme, the same parameters can be used for signing/verification as well as for decryption/encryption. The keys however are reversed, the public key is used to verify a signature and the private key to sign the message. In what follows we assume that a hash function is fixed for the signing and verification operations.

- ***Signing***: given the message m and the private key $Pv = (n, d)$, use the hash function to compute the hash of the message m as $H(m)$, then compute the signature as: $s = H(m)^d \bmod n$.
- ***Verification***: given the message m, the signature s and the public key $Pb = (n, e)$, use the hash function to compute the hash of the message m as $H(m)$ then verify that $H(m) = s^e \bmod n$.

RSA speed-up via CRT. In practical applications, computations are rarely performed modulo n, instead, they are done modulo the divisors of the modulus. This is achieved by following a result known as the Chinese Remaindering Theorem (CRT). Fix dp, dq by reducing the private exponent modulo $p-1$ and $q-1$. If we compute:

$$\begin{cases} m' = c^{dp} \bmod p \\ m'' = c^{dq} \bmod q \end{cases}$$

then the message m can be uniquely recovered modulo n by merging the two parts m', m'' as $m = \left(m'q(q^{-1} mod\ p) + m''p(p^{-1} mod\ q)\right) \bmod n$. This straight-forward solution was given by Gauss. It implies that exponentiation, which is the most intensive computational step, is done modulo the factors of the modulus which are usually half the bit-length of the modulus (e.g., for a 2048 bit modulus, you perform exponentiation over its 1024 factors). Another way to extract the message is by computing $m = m'' + q(q^{-1} mod\ p)(m' - m'') \bmod p$ which eliminates even the final computation modulo n (this final computation is in fact cheap compared to exponentiation). This trick is also used in .NET, a reason for which the private keys contain more than the modulus, public and private exponents. The full structure of the key will be detailed in a forthcoming section.

CCA security with padding. RSA is never used in practice without some padding of the plaintext. The padding assures that the cryptosystem is actually secure against active adversaries. The details for the padding scheme are too complex for this section (details should be given in a lecture that introduces some theoretical background). All you should know is that the padding adds a fixed form to the message which will disallow an adversary to manipulate a ciphertext such that it will correctly decrypt. In the simplest form, padding consists of simply appending some fixed 0x00 and 0xFF bytes to the message before encrypting it, e.g, encrypt the message as $c = (0\text{xFF}||0\text{xFF}||0\text{xFF}||m) \bmod n$ (here || denotes concatenation). In .NET two padding schemes are available, one is the secure OAEP padding (recommended) the other is a deprecated PKCS padding. How to set on of these will be discussed in the next section, details on the padding schemes are available in the lecture material.

4.2 RSACryptoServiceProvider: Properties and Methods

The RSA implementation in .NET supports keys from 384 to 16384 bits in 8 bit increments. The key size can be specified via the constructor of the *RSACryptoServiceProvider* class which will generate a random RSA key. The constructor also allows initialization with an existing key given as *CspParameters* object. In the forthcoming section we give more details on the key structure, now we will focus on the properties and methods exposed by the *RSACryptoServiceProvider* class, these are summarized in Tables 1 and 2. Figure 2 shows the class hierarchy for RSA and DSA in .NET.

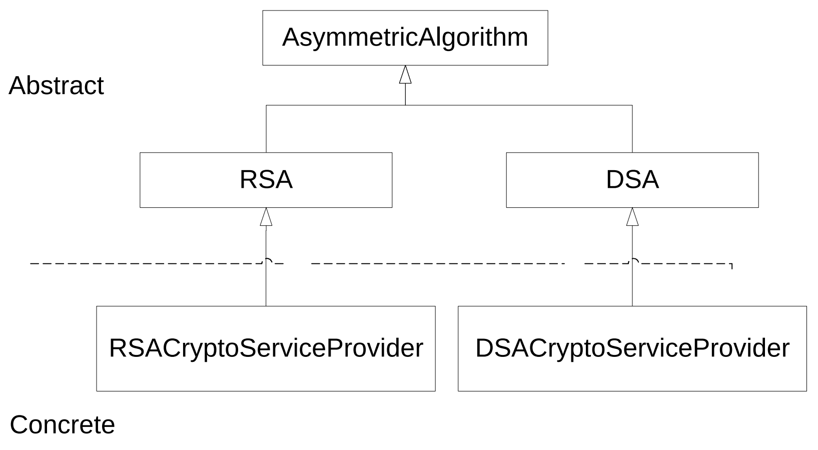


Figure 2. The RSA and DSA clases in .NET

	Get/Set	*Type*	**Brief Description**
PublicOnly	*g*	*Boolean*	Return true if the object contains just the public key
KeySize	*g*	*Int*	Key size in bits
LegalKeySizes	*g*	*KeySizes[]*	Key sizes in bits supported by the algorithm
SignatureAlgorithm	*g*	*String*	The name of the signature algorithm, in .NET signing is always performed as RSA with SHA1

Table 1. Properties from the RSACryptoServiceProvider class

	Return type	**Brief Description**
Decrypt (byte[] data, bool fOAEP)	*byte[]*	Decrypts data given as byte and returns the decrypted value as byte. The Boolean indicates if the OAEP padding is used, if false, then PKCS# v.15 padding is used instead.

Encrypt (byte[] data, bool fOAEP)	*byte[]*	Encrypts data given as byte and returns the encrypted value as byte. The Boolean indicates if the OAEP padding is used, if false, then PKCS# v.15 padding is used instead.
ExportParameters (bool includePrivateParameters)	*RSAParameters*	Gets the RSA key as RSAParameters object. The Boolean specifies if the private part of the key is or not included.
ImportParameters (RSAParameters parameters)	*void*	Sets the RSA key from RSAParameters object
ToXmlString (bool includePrivateParameters)	*string*	Gets the RSA key as string in XML format. The Boolean specifies if the private part of the key is or not included.
FromXmlString (bool includePrivateParameters)	*void*	Sets the RSA key from a string in XML format.
SignData (byte[] buffer, Object halg)	*byte[]*	Signs the given array of bytes with the specified hash algorithm, returns the signature as array of bytes
SignData(Stream inputStream, Object halg)	*byte[]*	Same as previously, but this time the data is given as stream
SignData(byte[] buffer, int offset, int count, Object halg)	*byte[]*	Signs the byte array starting from *offset* for *count* bytes
SignHash(byte[] hash, string str)	*byte[]*	Signs the hash of the data, the string is the name of the algorithm that was used to hash the data
VerifyData(byte[] buffer, Object halg, byte[] signature)	*bool*	Verifies the signature given a hash algorithm as object, the signature and message as byte arrays

VerifyHash (byte[] Hash, string str, byte[] Signature)	*bool*	Verifies the signature given the hash of the message and the name of the hash algorithm

Table 2. Methods from the RSACryptoServiceProvider class

Encryption and signing with RSA in .NET. Encryption and decryption with RSA should now be straight-forward. There are only two steps that you need to follow: i) create the RSA object (easiest way is by specifying the size of the key) and ii) call the encrypt method on the data specified as byte array and a Boolean which indicates if OAEP is to be used (recommended).

```
RSACryptoServiceProvider myRSA = new RSACryptoServiceProvider(2048);
AesManaged myAES = new AesManaged();
byte[] RSAciphertext;
byte[] plaintext;
//generate an AES key
myAES.GenerateKey();
//encrypt an AES key with RSA
RSAciphertext = myRSA.Encrypt(myAES.Key, true);
//decrypt and recover the AES key
plaintext = myRSA.Decrypt(RSAciphertext, true);
```

Table 3. Example of RSA encryption and decryption in .NET

```
SHA256Managed myHash = new SHA256Managed();
string some_text = "this is an important message";
//sign the message
byte[] signature;
signature =
myRSA.SignData(System.Text.Encoding.ASCII.GetBytes(some_text), myHash);
//verified a signature on a given message
bool verified;
```

```
verified =
myRSA.VerifyData(System.Text.Encoding.ASCII.GetBytes(some_text),
myHash, signature);
```

Table 4. Example of RSA signing and verification in .NET

4.3 The Structure of the Public and Private Key

The structure of the RSA key in .NET follows the PKCS #1 (Public Key Cryptography Standards) description. In contrast to the text-book description of the RSA scheme, the key includes parameters that are used to provide speed-ups with the Chinese Remaindering Theorem as discussed previously. The following parameters are present in the key:

- ✓ **Modulus** – the modulus, i.e., n,
- ✓ **Exponent** – the public exponent, i.e., e,
- ✓ **P** – the first prime factor of the modulus, i.e., p,
- ✓ **Q** – the second prime factor of the modulus, i.e., q,
- ✓ **DP** – the private exponent modulo p-1, i.e., $d \bmod p - 1$,
- ✓ **DQ** – the private exponent modulo q-1, i.e., $d \bmod q - 1$,
- ✓ **InverseQ** – the inverse of q modulo p, i.e., q^{-1} modp,
- ✓ **D** – the private exponent, i.e., d.

Exporting and importing keys as XML strings. The key can be exported to XML Strings with the methods *ToXMLString (bool includePrivateParameters)* which take a Boolean input specifying if the private part of the key is or not included in the returned string. In Tables 5 and 6 we show an RSA key exported from .NET with and without the private part. A key can also be imported from such a string via the *FromXmlString(string xmlString)* method.

<RSAKeyValue>
<Modulus>
uPmqM3pzkazPZAVC0pCA+unlLorxuxcwZb/AwcOE64qAIUZuLjRCKc0HFyJSwp38qw
y2JWNm7vQQmsm9xVECcBTUqTVR17hviNwof6qJ1BlpFbNqS5IXPM1oj2spVKVvaiC
nE+RPegQ2AZACxEOkoGZBxQFupfbbuzuoMNEt3qs=

```
</Modulus>
<Exponent>
AQAB
</Exponent>
<P>
/BP+eh9ZiAw5PXjniNzEEZ8+5+q12lYQ5peCJDUHNkzA7yyhWo9ayg+ZRt2yJ7tFvgF4t
RLF0nCBrJDQvSWTUw==
</P>
<Q>
u9pn4Ph7MDEAgwSk6lVrOe8mH3XW7f54l57LwklcBc7tzzB3DHsQ6UEJUfmTTE4ed5
ogX52F7hPVcXW86w40SQ==
</Q>
<DP>
KVBJk9BRhyehtf57zAWKqOy1jaL9HQSgDnrkXHTIctDPiiOBams2UQmPcHrjOPnLa2G
oW9zwyRWhWxv86hMfew==
</DP>
<DQ>
PQoPvPMgnB0gDHKC373XtKB3o7tXlkecia/Ih53sr9p4PV2DIWQPr6s5SxCsgxvTHIvRP
yBhN2XscgyO0VXxOQ==
</DQ>
<InverseQ>
pr2OyNnCyceTOWWPGn3x9yCHyPaAYiHyP/dLFNKqGmgWLkShtBbuVO8t97dtNPNd
sgHeS8mxnpZV0hxoYVJodQ==
</InverseQ>
<D>
qYzv3c/YLydf0iagYbHjCBts34Ssnvlae2mQngtBw0VovRd51xA/tWEhpqrngUyfVYqJSy
waJd3BeqCBOmRO/ipZRd4SXr3HX4vU3qtTwtSOKHMJW8BEn2dwgW3B4xbQkWo+t
i7VJZIxLSMS02IowLs3FfwjXz2ATVx71LqywoE=
</D>
</RSAKeyValue>
```

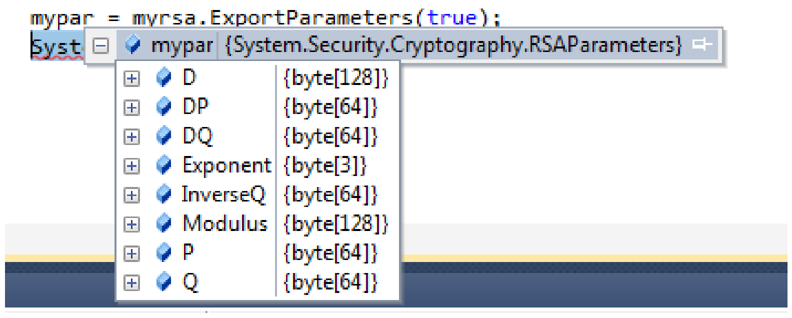
Table 5. RSA key exported as XML string with private parameters

```
<RSAKeyValue>
<Modulus>uPmqM3pzkazPZAVC0pCA+unlLorxuxcwZb/AwcOE64qAIUZuLjRCKc0HFy
JSwp38qwy2JWNm7vQQmsm9xVECcBTUqTVR17hviNwof6qJ1BlpFbNqS5IXPM1oj2s
pVKVvaiCnE+RPegQ2AZACxEOkoGZBxQFupfbbuzuoMNEt3qs=
</Modulus>
<Exponent>
AQAB
```

```
</Exponent>
</RSAKeyValue>
```

Table 6. RSA key exported as XML string without private parameters (just the public key)

Exporting and importing keys as byte arrays. Similarly, keys can be imported and exported as *System.Security.Cryptography.RSAParameters* which is a structure containing a byte array for each of the previously described parameters. This import/export method is needed when you want to import/export the key between distinct platforms, e.g., to a C++ or Java implementation. Figure 3 shows a screen capture from the .NET environment exposing the structure of a key.



```
mypar = myrsa.ExportParameters(true);
Syst
mypar {System.Security.Cryptography.RSAParameters}
  D         {byte[128]}
  DP        {byte[64]}
  DQ        {byte[64]}
  Exponent  {byte[3]}
  InverseQ  {byte[64]}
  Modulus   {byte[128]}
  P         {byte[64]}
  Q         {byte[64]}
```

Figure 3. Fields of an *RSAParameters* structure

4.4 EXERCISES

1. Evaluate the computational cost of RSA cryptosystem in .NET in terms of: key generation, encryption, decryption, signing and verification time. Results have to be presented in a tabular form as shown below.

1024 bit	2048 bit	3072 bit	4096 bit

Table 1. Cost of RSA key generation

1024 bit	2048 bit	3072 bit	4096 bit

Table 3. Cost of RSA encryption

1024 bit	2048 bit	3072 bit	4096 bit

Table 4. Cost of RSA dencryption

1024 bit	2048 bit	3072 bit	4096 bit

Table 5. Cost of RSA signing

1024 bit	2048 bit	3072 bit	4096 bit

Table 6. Cost of RSA verification

2. Given the data in the Table below, columns a) and b) are the modulus and private exponent for an RSA in .NET. The public exponent is the standard value 65537. Find the factorization of the modulus. In columns c) and d) are the modulus and dp parameter of an RSA object in .NET. Find the factorization of this modulus. *Note that all values are specified as byte arrays*.

(a)	(b)	(c)	(d)
m[0]=220;	d[0]=16;	m[0]=184;	dp[0]=66;
m[1]=94;	d[1]=158;	m[1]=180;	dp[1]=59;
m[2]=85;	d[2]=240;	m[2]=103;	dp[2]=152;
m[3]=235;	d[3]=222;	m[3]=69;	dp[3]=232;

m[4]=39;	d[4]=6;	m[4]=37;	dp[4]=217;
m[5]=74;	d[5]=157;	m[5]=247;	dp[5]=214;
m[6]=145;	d[6]=162;	m[6]=146;	dp[6]=230;
m[7]=228;	d[7]=57;	m[7]=23;	dp[7]=70;
m[8]=229;	d[8]=96;	m[8]=244;	dp[8]=190;
m[9]=175;	d[9]=117;	m[9]=94;	dp[9]=80;
m[10]=179;	d[10]=139;	m[10]=170;	dp[10]=43;
m[11]=77;	d[11]=17;	m[11]=104;	dp[11]=249;
m[12]=99;	d[12]=136;	m[12]=248;	dp[12]=24;
m[13]=158;	d[13]=53;	m[13]=128;	dp[13]=113;
m[14]=229;	d[14]=0;	m[14]=10;	dp[14]=93;
m[15]=79;	d[15]=216;	m[15]=221;	dp[15]=218;
m[16]=165;	d[16]=171;	m[16]=77;	dp[16]=69;
m[17]=70;	d[17]=255;	m[17]=32;	dp[17]=102;
m[18]=68;	d[18]=139;	m[18]=26;	dp[18]=135;
m[19]=238;	d[19]=205;	m[19]=31;	dp[19]=244;
m[20]=144;	d[20]=110;	m[20]=69;	dp[20]=252;
m[21]=203;	d[21]=144;	m[21]=153;	dp[21]=36;
m[22]=0;	d[22]=81;	m[22]=134;	dp[22]=161;
m[23]=1;	d[23]=20;	m[23]=148;	dp[23]=48;
m[24]=128;	d[24]=203;	m[24]=62;	dp[24]=179;
m[25]=140;	d[25]=236;	m[25]=43;	dp[25]=96;
m[26]=219;	d[26]=83;	m[26]=85;	dp[26]=172;
m[27]=107;	d[27]=212;	m[27]=241;	dp[27]=14;
m[28]=129;	d[28]=92;	m[28]=76;	dp[28]=136;
m[29]=46;	d[29]=238;	m[29]=12;	dp[29]=191;
m[30]=203;	d[30]=249;	m[30]=86;	dp[30]=23;
m[31]=3;	d[31]=146;	m[31]=178;	dp[31]=96;

m[32]=116;	d[32]=238;	m[32]=185;	dp[32]=33;
m[33]=99;	d[33]=33;	m[33]=160;	dp[33]=186;
m[34]=74;	d[34]=91;	m[34]=105;	dp[34]=226;
m[35]=45;	d[35]=3;	m[35]=45;	dp[35]=247;
m[36]=148;	d[36]=235;	m[36]=138;	dp[36]=78;
m[37]=89;	d[37]=15;	m[37]=239;	dp[37]=9;
m[38]=187;	d[38]=133;	m[38]=153;	dp[38]=24;
m[39]=113;	d[39]=138;	m[39]=224;	dp[39]=172;
m[40]=209;	d[40]=197;	m[40]=79;	dp[40]=143;
m[41]=50;	d[41]=27;	m[41]=122;	dp[41]=55;
m[42]=124;	d[42]=136;	m[42]=100;	dp[42]=238;
m[43]=34;	d[43]=175;	m[43]=205;	dp[43]=97;
m[44]=143;	d[44]=86;	m[44]=218;	dp[44]=247;
m[45]=173;	d[45]=164;	m[45]=253;	dp[45]=44;
m[46]=248;	d[46]=233;	m[46]=120;	dp[46]=251;
m[47]=137;	d[47]=124;	m[47]=16;	dp[47]=235;
m[48]=69;	d[48]=249;	m[48]=201;	dp[48]=237;
m[49]=229;	d[49]=15;	m[49]=83;	dp[49]=86;
m[50]=3;	d[50]=151;	m[50]=187;	dp[50]=165;
m[51]=114;	d[51]=221;	m[51]=8;	dp[51]=252;
m[52]=121;	d[52]=247;	m[52]=91;	dp[52]=58;
m[53]=67;	d[53]=117;	m[53]=31;	dp[53]=187;
m[54]=207;	d[54]=124;	m[54]=175;	dp[54]=77;
m[55]=85;	d[55]=218;	m[55]=8;	dp[55]=247;
m[56]=57;	d[56]=209;	m[56]=36;	dp[56]=254;
m[57]=252;	d[57]=191;	m[57]=217;	dp[57]=128;
m[58]=222;	d[58]=215;	m[58]=104;	dp[58]=98;
m[59]=148;	d[59]=205;	m[59]=18;	dp[59]=88;

m[60]=56;	d[60]=216;	m[60]=201;	dp[60]=10;
m[61]=188;	d[61]=37;	m[61]=84;	dp[61]=86;
m[62]=167;	d[62]=170;	m[62]=118;	dp[62]=190;
m[63]=127;	d[63]=128;	m[63]=60;	dp[63]=245;
m[64]=109;	d[64]=87;	m[64]=178;	
m[65]=174;	d[65]=22;	m[65]=120;	
m[66]=208;	d[66]=90;	m[66]=147;	
m[67]=247;	d[67]=156;	m[67]=150;	
m[68]=63;	d[68]=150;	m[68]=55;	
m[69]=201;	d[69]=185;	m[69]=110;	
m[70]=145;	d[70]=12;	m[70]=14;	
m[71]=150;	d[71]=105;	m[71]=185;	
m[72]=188;	d[72]=247;	m[72]=237;	
m[73]=99;	d[73]=207;	m[73]=127;	
m[74]=162;	d[74]=244;	m[74]=212;	
m[75]=246;	d[75]=226;	m[75]=204;	
m[76]=54;	d[76]=154;	m[76]=8;	
m[77]=72;	d[77]=247;	m[77]=72;	
m[78]=51;	d[78]=179;	m[78]=92;	
m[79]=219;	d[79]=186;	m[79]=191;	
m[80]=198;	d[80]=162;	m[80]=136;	
m[81]=43;	d[81]=18;	m[81]=34;	
m[82]=186;	d[82]=162;	m[82]=74;	
m[83]=166;	d[83]=232;	m[83]=232;	
m[84]=162;	d[84]=175;	m[84]=130;	
m[85]=7;	d[85]=169;	m[85]=123;	
m[86]=115;	d[86]=72;	m[86]=10;	
m[87]=137;	d[87]=50;	m[87]=91;	

m[88]=105;	d[88]=64;	m[88]=123;	
m[89]=164;	d[89]=7;	m[89]=229;	
m[90]=248;	d[90]=232;	m[90]=254;	
m[91]=20;	d[91]=92;	m[91]=231;	
m[92]=201;	d[92]=117;	m[92]=209;	
m[93]=164;	d[93]=99;	m[93]=67;	
m[94]=159;	d[94]=8;	m[94]=231;	
m[95]=140;	d[95]=118;	m[95]=74;	
m[96]=4;	d[96]=32;	m[96]=220;	
m[97]=202;	d[97]=108;	m[97]=137;	
m[98]=52;	d[98]=133;	m[98]=195;	
m[99]=106;	d[99]=164;	m[99]=169;	
m[100]=126;	d[100]=23;	m[100]=186;	
m[101]=94;	d[101]=174;	m[101]=35;	
m[102]=87;	d[102]=111;	m[102]=139;	
m[103]=55;	d[103]=100;	m[103]=32;	
m[104]=168;	d[104]=94;	m[104]=222;	
m[105]=176;	d[105]=225;	m[105]=137;	
m[106]=137;	d[106]=202;	m[106]=40;	
m[107]=114;	d[107]=182;	m[107]=211;	
m[108]=157;	d[108]=59;	m[108]=158;	
m[109]=3;	d[109]=116;	m[109]=155;	
m[110]=111;	d[110]=6;	m[110]=30;	
m[111]=213;	d[111]=16;	m[111]=249;	
m[112]=134;	d[112]=112;	m[112]=63;	
m[113]=40;	d[113]=53;	m[113]=100;	
m[114]=155;	d[114]=215;	m[114]=10;	
m[115]=74;	d[115]=173;	m[115]=167;	

m[116]=81;	d[116]=16;	m[116]=87;	
m[117]=68;	d[117]=220;	m[117]=92;	
m[118]=171;	d[118]=117;	m[118]=107;	
m[119]=47;	d[119]=141;	m[119]=190;	
m[120]=129;	d[120]=35;	m[120]=251;	
m[121]=160;	d[121]=107;	m[121]=111;	
m[122]=210;	d[122]=240;	m[122]=199;	
m[123]=34;	d[123]=110;	m[123]=177;	
m[124]=240;	d[124]=195;	m[124]=13;	
m[125]=38;	d[125]=136;	m[125]=212;	
m[126]=168;	d[126]=209;	m[126]=194;	
m[127]=211;	d[127]=113;	m[127]=9;	

Note: You may consider recycling the code below

```
RSACryptoServiceProvider myrsa = new RSACryptoServiceProvider(1600);
            System.Diagnostics.Stopwatch swatch = new
System.Diagnostics.Stopwatch();
int size;
int count = 100;
swatch.Start();
for (int i = 0; i < count; i++)
{
  myrsa = new RSACryptoServiceProvider(2048);
  size = myrsa.KeySize;
}
swatch.Stop();
Console.WriteLine("Generation time at 1024 bit ... " +
(swatch.ElapsedTicks / (10*count)).ToString() + " ms");
byte[] plain = new byte[20];
byte[] ciphertext = myrsa.Encrypt(plain, true);

swatch.Reset();
swatch.Start();
```

```
for (int i = 0; i < count; i++)
{
    ciphertext = myrsa.Encrypt(plain, true);
}
swatch.Stop();
Console.WriteLine("Encryption time at 1024 bit ... " +
(swatch.ElapsedTicks / (10 * count)).ToString() + " ms");

swatch.Reset();
swatch.Start();
for (int i = 0; i < count; i++)
{
      plain = myrsa.Decrypt(ciphertext, true);
}
swatch.Stop();
Console.WriteLine("Decryption time at 1024 bit ... " +
(swatch.ElapsedTicks / (10 * count)).ToString() + " ms");

Console.ReadKey();
```

Chapter 5. THE DSA SIGNATURE ALGORITHM IN .NET

This section presents the DSA (Digital Signature Algorithm) as implemented in .NET. The .NET framework contains two digital signature schemes, the RSA which was previously discussed and DSA, also known as DSS (Digital Signature Standard). The DSA is a discrete logarithm based signature scheme, based on the ElGamal signature, which was standardized by NIST.

5.1 BRIEF THEORETICAL BACKGROUND

You are referred to the lecture material for more details on the DSA. However, to make things clearer, we do a brief recap on how DSA works. The details of this algorithm are more complicated than for the RSA, in particular the details are somewhat uneasy to memorize as the construction appears less natural than the RSA. The straight-forward idea of using the encryption key for verification and the decryption key for signing does not work anymore, however, the algorithm is in fact slightly more efficient and secure than the RSA and not hard to implement (it is based on the same core operation: modular exponentiation).

How the DSA signature works. As any public key signature, the DSA is a collection of three algorithms:

- ***Key generation:*** generate a random prime p and a second random prime q such that it divides $p-1$ then select an element g of Z_p of order q and a random number a from Z_p. The public key is $Pb = (g, g^a \bmod p, p)$ and the private key is $Pv = (g, a, p)$ **(q is fixed at 160 for the .NET implementation due to the use of SHA1)**
- ***Signing***: given the message m, use the hash function (SHA1 in .NET) to compute the hash of the message h, then select a random $k \in (0, p-1)$ and compute $r = g^k \bmod p$ then $s = k^{-1}(\mathrm{h} + \mathrm{ar}) \bmod (p-1)$. The signature is the pair (r, s).
- ***Verification***: given the signature (r, s), first check that $r \in (0, p)$, $s \in (0, q)$ (if not the signature is considered false) otherwise verify that $v = r$ where $v = g^{u_1} y^{u_2}, u_1 = wh \bmod q, u_2 = rw \bmod q, w = s^{-1}$ and return true if this holds (otherwise the signature is considered false).

Fortunately, you do not need to remember all these details in order to use this signature scheme in .NET, all you have to do is to call the methods for signing and verification. We discuss these next.

5.2 DSACryptoServiceProvider: Properties and Methods

The DSA implementation in .NET supports keys from 512 to 1024 bits in 64 bit increments. The key size can be specified via the constructor of the *DSACryptoServiceProvider* class which will generate a random DSA key. Similar to the case of the RSA, the constructor also allows initialization with an existing key given as *CspParameters* object. However, the methods for signing and verification do not offer the possibility of using an external hash object, in .NET this signature is bound to SHA1. These methods are summarized in Table 1, the distinction with the RSA is the absence of the hash algorithm as parameter since this is implicitly set to SHA1. The *DSACryptoServiceProvider* also has an additional *VerifySignature* method that takes the hash and signature of the message as input.

	Return type	**Brief Description**
ExportParameters (bool includePrivateParameters)	*DSAParameters*	Gets the DSA key as RSAParameters object. The Boolean specifies if the private part of the key is or not included.
ImportParameters (DSAParameters parameters)	*void*	Sets the DSA key from DSAParameters object
ToXmlString (bool includePrivateParameters)	*string*	Gets the DSA key as string in XML format. The Boolean specifies if the private part of the key is or not included.
FromXmlString (bool includePrivateParameters)	*void*	Sets the DSA key from a string in XML format.
SignData(byte[] buffer)	*byte[]*	Signs the given array of bytes with the specified hash algorithm, returns the signature as array of bytes

SignData(Stream inputStream)	*byte[]*	Same as previously, but this time the data is given as stream
SignData(byte[] buffer, int offset, int count)	*byte[]*	Signs the byte array starting from *offset* for *count* bytes
SignHash(byte[] hash, string str)	*byte[]*	Signs the hash of the data, the string is the name of the algorithm that was used to hash the data
VerifyData(byte[] buffer, byte[] signature)	*bool*	Verifies the signature given a hash algorithm as object, the signature and message as byte arrays
VerifyHash (byte[] Hash, string str, byte[] Signature)	*bool*	Verifies the signature given the hash of the message and the name of the hash algorithm
VerifySignature(byte[] Hash, byte[] Signature)	*bool*	Verifies the signature given the hash of the message

Table 1. Methods from the *DSACryptoServiceProvider* class

Signing with DSA in .NET. Signing requires the instantiation of a *DSACryptoServiceProvider* object and then calling one of the signing methods, same for verification. This is suggested in the code from Table 2.

```
DSACryptoServiceProvider myDSA = new DSACryptoServiceProvider(512);
byte[] sig = myDSA.SignData(data);
bool verify = myDSA.VerifyData(data, sig);
```

Table 2. Example for signing a byte array and verifying the signature in .NET with DSA

5.3 The Structure of the Public and Private Key

We now enumerate the parameters of the DSA private and public key:

- ✓ **P** – the prime that defines the group, i.e., p,
- ✓ **Q** – the factor of p-1 which gives the order of the subgroup, i.e., q, (this is always 160 bits in .NET)
- ✓ **G** – the generator of the group, i.e., g,
- ✓ **Y** – the value of the generator to X, i.e., $y = g^x \bmod p$
- ✓ **J** – a parameter specifying the quotient from dividing p-1 to q, i.e., $j = (p - 1)/q$,
- ✓ **Seed** – specifies the seed used for parameter generation,
- ✓ **Counter** – a counter value that results from the parameter generation process,
- ✓ **X** – a random integer, this is the secret part of the key, i.e., parameter a from the description of the scheme.

Exporting and importing keys as XML strings. Similar to the case of RSA the key can be exported to (or imported from) XML Strings. In Tables 3 and 4 we show a DSA key exported from .NET with and without the private part.

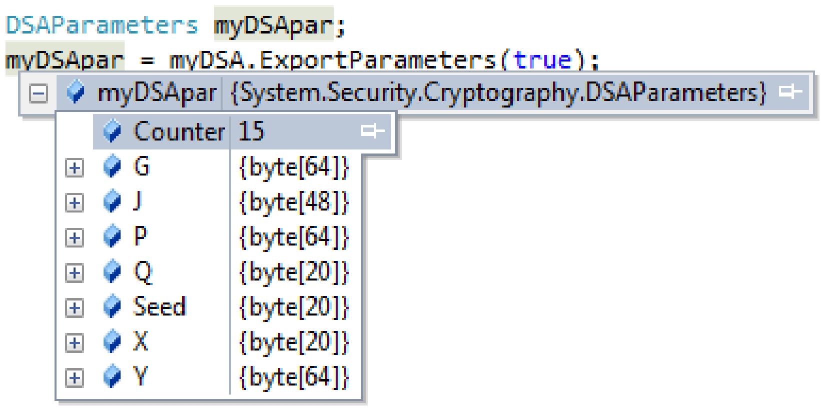
```
<DSAKeyValue>
<P>
sRp/2qfasQ+6ObB/6+7HqyZnmgp0drn7G/ewLihzFfiJrVS15Wu5slPXYY8lIpiqbwgVWj
5UMoV1ynnmx392YQ==
</P>
<Q>
+DqDOhkIdeiQrtipZf6d/ei35Yc=
</Q>
<G>
NElPMJMiLsqzHWyFmQeLNESbdmRNTta78aApURYyCqZ9CVTQCZTwX/N5YpulkCKG
KwOxMkXRdfAB0XVDQj/nJQ==
</G>
<Y>
KYtWDqa9aRI/bP5q82sfpSutSWJqDnkS9INGhZbdHxHcJw4XMU/ihIHUzS3zkODneM
nj3kz0Ly3jMJvkcm15kw==
</Y>
<J>
tqXwIvpqvwSLuUWWfcrGaUyl9AP07V0qfib1UtBD2xyf0c9sHacjniyQbqA=
</J>
<Seed>
nVHH51WY6AQqRGBXYDg+zQnHF5s=
</Seed>
<PgenCounter>
Ag==
</PgenCounter>
<X>
NhAM2W6d+VISPUZ967Zfc8v8D+8=
</X>
</DSAKeyValue>
```

Table 3. DSA key exported as XML string with private parameters

```
<DSAKeyValue>
<P>
sRp/2qfasQ+6ObB/6+7HqyZnmgp0drn7G/ewLihzFfiJrVS15Wu5slPXYY8lIpiqbwgVWj
5UMoV1ynnmx392YQ==
</P>
<Q>
+DqDOhkIdeiQrtipZf6d/ei35Yc=
</Q>
<G>
NElPMJMiLsqzHWyFmQeLNESbdmRNTta78aApURYyCqZ9CVTQCZTwX/N5YpulkCKG
KwOxMkXRdfAB0XVDQj/nJQ==
</G>
<Y>
KYtWDqa9aRI/bP5q82sfpSutSWJqDnkS9INGhZbdHxHcJw4XMU/ihIHUzS3zkODneM
nj3kz0Ly3jMJvkcm15kw==
</Y>
<J>
tqXwIvpqvwSLuUWWfcrGaUyl9AP07V0qfib1UtBD2xyf0c9sHacjniyQbqA=
</J>
<Seed>
nVHH51WY6AQqRGBXYDg+zQnHF5s=
</Seed>
<PgenCounter>
Ag==
</PgenCounter>
</DSAKeyValue>
```

Table 4. DSA key exported as XML string without private parameters (just the public key)

Exporting and importing keys as byte arrays. Identical to the case of RSA, keys can be imported and exported as *System.Security.Cryptography.DSAParameters* which is a structure containing a byte array for each of the previously described parameters. This is suggested in Figure 1.



```
DSAParameters myDSApar;
myDSApar = myDSA.ExportParameters(true);
myDSApar {System.Security.Cryptography.DSAParameters}
    Counter  15
    G        {byte[64]}
    J        {byte[48]}
    P        {byte[64]}
    Q        {byte[20]}
    Seed     {byte[20]}
    X        {byte[20]}
    Y        {byte[64]}
```

Figure 1. Fields of the *DSAParameters* structure

5.4 EXERCISES

2. Evaluate the computational cost of DSA signature in .NET in terms of: key generation, signing and verification time. Results have to be presented in a tabular form as shown below.

512 bit	640 bit	768 bit	1024 bit

Table 5. Cost of DSA key generation

512 bit	640 bit	768 bit	1024 bit

Table 6. Cost of DSA signing

512 bit	640 bit	768 bit	1024 bit

Table 7. Cost of DSA verification