Cryptography,

Application Notes in C, .NET and Java

Bogdan Groza

2015

Scope of the material

This material is mainly intended for students following lectures on Cryptography and Systems Security at Politehnica University of Timisoara (UPT), Romania. The main intention of these notes is to show that the theoretical objects discussed during lectures, besides their practical value which reasonably follows from relevance of information security today, are also present in a large variety of programming frameworks.

It was a main intention not to bind the content of the notes with a particular programming framework as cryptography is platform independent. For this reason, we make use of C programming under Linux (Section 1), .NET (Sections 2-5) and Java (Sections 6 and 7).

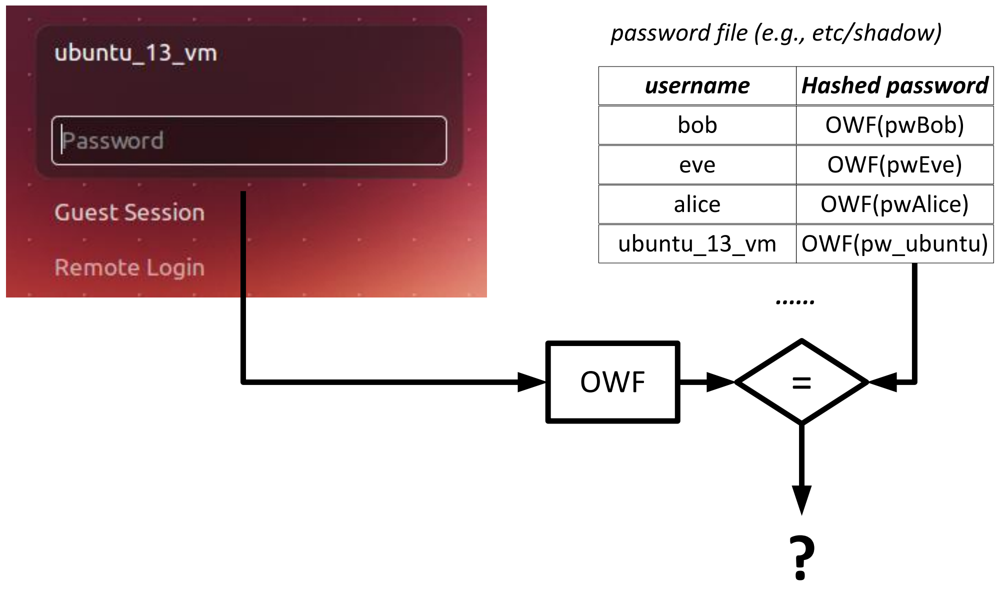
These notes are intended for engineers and are not focused on the design of cryptographic primitives which is a more demanding task, the material requires no background in cryptography.

CONTENTS

Chapter 1. THE UNIX PASSWORD BASED AUTHENTICATION SYSTEM

This chapter is centred on a simple but relevant subject: password based authentication (PBA). Regardless of the system, be it UNIX based, Windows, or even a remote system requiring PBA, e.g., on-line networks such as Facebook, LinkedIn, the paradigm is almost always the same: *the users enters a password which is verified against an encrypted version of the password that is stored locally on the system*. This encrypted version of the password is not always the result of applying an encryption function on the password, but rather applying some cryptographic one-way function (OWF). An OWF is a function that is easy to apply on the password but from which it is computationally infeasible to find the password, i.e., computing from input to output is easy while from output to input infeasible. Any cryptographic primitive can be used: hash functions, encryption functions, etc., since all these cryptographic primitives are OWFs. These functions will allow only for a random looking sequence to be stored in the password file, from which it should not be easy (or hopefully impossible) to guess the password of the user. Since usually hash functions (not encryption functions) are used for this purpose, we will refer to this encrypted value of the password as hashed password (note however that an encryption function such as DES or Blowfish can be used for the same purpose, in fact these are ready to use alternatives in most Linux distributions despite the more common use of MD5 or SHA2). If you are not yet familiar with hash functions, all that you should know for the moment is that they are OWFs that takes as input a string of any length and turns it into a value of fixed size, e.g., 128 bits in case of MD5, 160 bits in case of SHA1, 256 bits in case of SHA256, etc. that is usually referred as tag or hash.

The necessity for encrypting the passwords before storing them comes from the need of protecting one user from another (usually from admins or super-users) that can snitch on the password file (this is usually the case for super-users). Indeed this protection is not perfect, one can plant a key-logger and record all user input, install a Trojan that records activity at login, etc. However, if we assume that the system is clean from such malicious objects (and this is a reasonable assumption in many situations), then the best one could do is to read the file in which the passwords are stored. Consequently, encrypting the passwords is a good security decision.



password file (e.g., etc/shadow)

username	***Hashed password***
bob	OWF(pwBob)
eve	OWF(pwEve)
alice	OWF(pwAlice)
ubuntu_13_vm	OWF(pw_ubuntu)

……

Figure 1. Password based authentication in Ubuntu

The way in which passwords are encrypted varies from one system to another, here we focus on how this is done under UNIX (and in particular the Ubuntu OS which we assume to be installed on your computer). The user authentication works in a straight-forward way: when the user enters his password at the login screen, the password is passed through a one-way function (the same which was used when it was stored) and the output is verified against the value stored in this passwords file. If the values are identical the users gains access, otherwise it is rejected (usually there is only a limited number of attempts and there is some delay after entering a wrong password in order to prevent attacks). This mechanism is suggested in Figure 1.

1.1 THE PASSWD AND SHADOW FILES

Traditionally, in UNIX based operating systems the hashed passwords were stored in the file */etc/passwd* (a text file). On almost all recent distributions (including Ubuntu 13 which we assume to be deployed on your computer) the *passwd* file contains only some user related information while the hashed passwords are not here but in the */etc/shadow* file (also a text file, but with limited access, e.g., it cannot be accessed by regular users). This is done in order to increase security by disallowing regular users from reading it. The *passwd* file can be accessed by all users in read mode, however the *shadow* file is accessible only to super-users.

Adding users and passwords. To play a bit with the *password* and *shadow* files we first add some users, say *tom*, *alice* and *bob*. To add users use the command *sudo useradd –m username* (-m creates the home directory of the user) then to set the password use *sudo passwd username* (*sudo* allows you to run the *usearadd* and *passwd* commands with super-user privileges). If you need help on any of this commands use *man useradd* or *man passwd*.

```
ubuntu@ubuntu:~$ sudo useradd -m tom
ubuntu@ubuntu:~$ sudo passwd tom
Enter new UNIX password:
Retype new UNIX password:
```

Table 1. Creating a user named tom and setting his password

Accessing the shadow file. To access the *shadow* file you also need super-user privileges, for this, in the terminal run *sudo gedit* and open the file from *gedit*. If you successfully managed to create these accounts then the *passwd* and *shadow* files should look similar to what you can see in Tables 2 and 3 (note the user names and their hashed passwords).

```
ubuntu:x:1000:1000:ubuntu_13_vm,,,:/home/ubuntu:/bin/bash
tom:x:1001:1001::/home/tom:/bin/sh
alice:x:1002:1002::/home/alice:/bin/sh
bob:x:1003:1003::/home/bob:/bin/sh
```

Table 2. Example of *passwd* file with 4 users: ubuntu, tom, alice and bob

```
ubuntu:$1$js9ai3VX$iFbR5uTfv3JMmFCladdcn1:16459:0:99999:7:::
tom:$6$vIkXOyrz$CMiFB8meMfTANianaS7z5f8yMfplk/TtncZs/7b0es65XZSIyz3k
aiSwN/61sBdrPhT9B0RulJ9tWEnE7kpJC/:16470:0:99999:7:::
alice:$6$gpOJXcSy$AVrdUKBdSM8NlGmrbexoyetS2LhRgg3qkaTbZdMh4mj.Yps3
UxIkrtGDQfEGA.yNDhlIPG3m1hupX3b0I0Vs3.:16470:0:99999:7:::
bob:$6$5IPGOooA$J6rZ74NUpCVx9C2mpesKKr0iBjkHNCxz8Io3aPj5W6mwVKKv
nhWIbq0H91T9bDcWmDE3/6Ageoa3olcVe2nKY0:16470:0:99999:7:::
```

Table 3. Example of *shadow* file with 4 users: ubuntu, tom, alice and bob

Structure of the passwd and shadow files. In the *passwd* file, the first field is the user name, while the x indicates that the passwords are not here but in the *shadow* file. Subsequently you can see the user identifier, group identifier, the user full name (and other potential information such as phone number, contact details, etc.), the home directory and the program that is started at login. The *shadow* file contains the information that is more relevant to us. Note the "$" sign in this file. Following the user name, in the *shadow* file, we have a *id* field which identifies the particular algorithm used to encrypt/hash the passwords. The following options are supported in your Ubuntu distribution:

- *1* - a version based on MD5 which is a hash function with 128 bit output that is no longer cryptographically secure (more details available in the lectures) but can still be somewhat safely used for this purpose,
- *$2a$* - Blowfish, a symmetric encryption algorithm, but not a usual option for this purpose,
- *5* - SHA-256 a hash function with 256 bit output,
- *6* - SHA-512 a hash function with 512 bit output which should give the maximum level of security.

After the algorithm identifier a random value *$salt$* follows. This value is called *salt* and is a randomly generated value, non-secret, that is used to prevent pre-computed attacks, i.e., you cannot compute the hash over a dictionary of passwords in an off-line manner since you do not know the salt and all your off-line computations will be of no use for a distinct salt value (it also prevents two users with the same password from having the same hash value in the *shadow* file). Finally, the *$hash$* value is the actual hash of the password. Other fields follow but not of much importance for this work: days since last change, days until change allow, days before change required, days warning for expiration, days before account inactive, days since epoch when account expires.

1.2 VERIFYING PASSWORDS PROGRAMMATICALLY

To generate the hash of a password, the *crypt()* function must be used. This function takes the password and the salt as character arrays, i.e., *char **, and returns a character array which is the hash of the password. The id in the salt dictates the particular algorithm that is to be used. This function can be called from any C/C++ program, but usually you will have to include *crypt.h* in order to work.

```
char *crypt(const char *key, const char *salt);
```

Table 4. The UNIX crypt function

Programs that use this function must be linked with the *–lcrypt* option, the sequence for compiling and running the program is in Table 5. Note that we assume the program test.cpp to be in the current directory and we specify the output file as *test* then run this file with *./test*.

```
ubuntu@ubuntu:/mnt/hgfs/VM_Shared$ g++ -o test test.cpp –lcrypt
ubuntu@ubuntu:/mnt/hgfs/VM_Shared$ ./test
```

Table 5. Compiling and running the program

1.3 EXHAUSTIVE SEARCH, A TRIVIAL ATTEMPT

Various programs for cracking passwords exist, but the purpose of this assignment is to help you in building your own. The program in Table 6 performs an exhaustive search for passwords of length at most MAX_LEN where the characters are chosen from a predefined set *char* charset.* How the code works should easily follow from the comments. The main idea is that we test each password that is generated by passing it through *crypt*, see *int check_password(char* pw, char* salt, char* hash)*. To generate all possible passwords from the predefined character set, i.e., *charset*, we take passwords of 1 character at the beginning and gradually apply to them each possible character, etc. All this is done inside *char* exhaustive_search(char* charset, char* salt, char* target)*.

```
#include <iostream>
#include <list>
#include <cstring>
#include <crypt.h>

using namespace std;

//this is an example line from the shadow file:
```

```
//$6$Iy/hHRfM$gC.Fw7CbqG.Qc9p9X59Tmo5uEHCf0ZAKCsPZuiYUKcejrsGu
ZtES1VQiusSTen0NRUPYN0v1z76PwX2G2.v1l1:15001:0:99999:7:::
// the salt and password values are extracted as

string target_salt = "$6$Iy/hHRfM$";
string target_pw_hash =
"$6$Iy/hHRfM$gC.Fw7CbqG.Qc9p9X59Tmo5uEHCf0ZAKCsPZui
YUKcejrsGuZtES1VQiusSTen0NRUPYN0v1z76PwX2G2.v1l1";

// define a null string which is returned in case of failure to find the password
char null[] = {'\0'};

// define the maximum length for the password to be searched
#define MAX_LEN 6

list<char*> pwlist;

// check if the pw and salt are matching the hash
int check_password(char* pw, char* salt, char* hash)
{
char* res = crypt(pw, salt);
cout << "password " << pw << "\n";
cout << "hashes to " << res << "\n";
for (int i = 0; i<strlen(hash); i++)
 if (res[i]!=hash[i]) return 0;
cout << "match !!!" << "\n";
return 1;
}

// builds passwords from the given character set
// and verifies if they match the target
char* exhaustive_search(char* charset, char* salt, char* target)
{
char* current_password;
char* new_password;
int i, current_len;

// begin by adding each character as a potential 1 character password
for (i = 0; i<strlen(charset); i++){
      new_password = new char[2];
 new_password[0] = charset[i];
```

```
new_password[1] = '\0';
pwlist.push_back(new_password);
}

while(true){

// test if queue is not empty and return null if so
if (pwlist.empty()) return null;

// get the current current_password from queue
current_password = pwlist.front();
current_len = strlen(current_password);

// check if current password is the target password, if yes return the
current_password
if (check_password(current_password, salt, target)) return
current_password;

// else generates new passwords from the current one by appending each
character from the charlist
// only if the current length is less than the maxlength
if(current_len < MAX_LEN){
for (i = 0; i < strlen(charset); i++){
        new_password = new char[current_len + 2];
        memcpy(new_password, current_password, current_len);
        new_password[current_len] = charset[i];
        new_password[current_len+1] = '\0';
        pwlist.push_back(new_password);
}
}
// now remove the front element as it didn't match the password
pwlist.pop_front();
}
}

main()
{
 char* salt;
 char* target;
 char* password;
 // define the character set from which the password will be built
```

```
 char charset[] = {'b', 'o', 'g', 'd', 'a', 'n', '\0'};
 //convert the salt from string to char*
 salt = new char[target_salt.length()+1];
 copy(target_salt.begin(), target_salt.end(), salt);
 //convert the hash from string to char*
 target = new char[target_pw_hash.length()+1];
 copy(target_pw_hash.begin(), target_pw_hash.end(), target);
 //start the search
 password = exhaustive_search(charset, salt, target);
 if (strlen(password)!= 0) cout << "Password successfuly recovered: " <<
password << "  \n";
 else cout << "Failure to find password, try distinct character set of size \n";
}
```

Table 5. An exhaustive search algorithm for finding the password.

1.4 EXERCISES

1. Consider passwords of 20 characters and that they are hashed through MD5 which outputs 128 bits. How many passwords of 20 characters are there for a single 128 bit output? How many users should be expected until a collision occurs with probability ½ ? (note that since hash functions are collision resistant, it is actually computationally infeasible to find such passwords, but it is good to understand that they do exist)

2. Find the password that corresponds to the following shadows entry, having in mind that the character set is {a, b, c, 1, 2, !, @, #} and the non-alphanumerical symbols occur only at the end of the password

```
tom:$6$SvT3dVpN$lwb3GViLl0J0ntNk5BAWe2WtkbjSBMXtSkDCtZUkVhVPiz5
X37WflWL4k3ZUusdoyh7IOUlSXE1jUHxIrg29p.:16471:0:99999:7:::
```

3. Consider a 14 character password that ranges over all possible ASCII symbols. On your current computer, how much time will you need to break such a password?

4. Consider the same context as previously, but this time we are concerned with memory usage. Could you provide a rough estimation of the amount of memory that is used to break the password in the previous example? Can you implement a solution that improves on this amount?

5. The following *shadow* entry was generated by a password formed by an arbitrary arrangement of the following words: *red*, *green*, *blue*, *orange*, *pink*. Find the password.

```
tom:$6$9kfonWC7$gzqmM9xD7V3zzZDo.3Fb5mAdM0GbIR2DYTtjYpcGkXVWat
TC0pa/XVvKTXLb1ZP0NG9cinGRZF7gPLdhJsHDM/:16471:0:99999:7:::
```

6. Now a more demanding exercise. All of the following passwords start with)):@$*!:((and the rules defined below for each user apply only for the predefined character set:

   Alpha = { a, b, c …, x, y, z } [1]

   Num = {0, 1, 2, …, 9}

   Sym = {!,@,#,$,%,^,&,*,(,)}

   a. ***tom_easy*** has a password from all characters in *Alpha*, *Num* and *Sym*, which gives a total of: 26 letters, 10 numbers and 10 symbols, summing up to 46 characters. The password contains at most 4 such characters, i.e., 46^4 = 4,477,456. [2]

   b. ***tom_harder*** has a password constructed from the same set _Alpha x Num x Sym_ except that after the starting characters “)):@$*!:((” it has an additional number from 1..10. Suggestion: to solve this, you may consider running 10 instances of the previous program with passwords starting with “)):@$*!:((1”, “)):@$*!:((2”, “)):@$*!:((3”, “)):@$*!:((4” and “)):@$*!:((5”,

---

[1] Note that there are no upper-case letters

[2] You should be able to crack this in ~12 hours (assuming that your computer can perform 5x10^6 passwords/day, check the exact running time with the *time* command)

etc. Since only the first solver gets the points, you may consider running these on distinct computers.

c. ***tom_split*** – has the first 4 characters from *Alpha* and the last 2 from *Num* & *Sym*. Suggestion, you should search separately for the first 4 and last 2 chars.
d. ***tom_wordy*** – has a concatenation in some random order of the following 8 words {the, big, brown, fox, or, small, grey, elephant}. Words may repeat but there are only 8 words.
e. ***tom_wordy_harder*** – has a concatenation in some random order of the following 10 words {the, big, brown, fox, or, small, gray, elephant, yesterday, today}. Words do not repeat.
f. ***tom_math*** – has a password of the form “)):@$*!:(($N_1N_2N_3N_4$“ where N_i is a number generated $N_i = N_{i-1} +$ *seed* **mod** 255 where N_0 and *seed* are random values in {0, 255}. The numbers are written as characters, i.e., if N1 = 234 then password is “)):@$*!:((234 … “
g. ***tom_more_math*** – same as previously but the operation is performed modulo 4096, as well as the seed and N_0 and *seed* are random values in {0, 4096}

Note: remember passwords start with:)):@$*!:((

```
tom_easy:$6$JcQryNT4$Nydvd9w9kpwwkTTU93uMulS9noTyiLmheUnyNrVaNoVjA
yyFAAXAXP1.EePMdYlohOyVAxcuplfZMQD7VixY7/:16497:0:99999:7:::

tom_hard:$6$tamx8Uvr$tMfa8QsdrJnDa6n40tVVy7kRaFbbgevr4rFz/rFNDTmaUcKn
ZiiBSGVkO/uS1/M513Z0BVuBELrhDrwr9EJRY0:16497:0:99999:7:::

tom_split:$6$Z9VfBmUG$MhG7XlzZnBxdgRjDf1utb7fJZSc8hvzPJhCjcBd.lN.HoMvsG
T1.wn0ACl.AydYq5oVw9uFCTtpH4oOa1s/bT1:16497:0:99999:7:::

tom_wordy:$6$GHuikUus$T8/C1Ed6QLBkHhMWJB/nFPgY/tujpMqkpy8tmG.ovijy96
0HrWSkPQWU8h062SuR/NIDIJhCbszMIycdILr7p1:16497:0:99999:7:::

tom_wordy_harder:$6$p0sCvjGG$ZH9..sdjHFaWux9lgVWm44USpVawFheB8I4PJcA
7ep9nj6ISwcCbO7/SvuTS9LdreyMO./zFPKyE06zR5G/Bw1:16497:0:99999:7:::
```

```
tom_math:$6$SMF7niTS$HuLhlRyIAnhLhNRtqqd/OSkye3fEsnd9i2trxx53Mji/hYZQ8
ywnIiUMa6hgSax/SOeCYTootE649Zzblt4Fq1:16497:0:99999:7:::

tom_math_harder:$6$/agnX0ga$kY0EejIuThPUH/DeTYJZIAPzxMA3WXYZjHOF/YKQ
a6jEM9lHNAKt9fRyVWGntpG/BPH3sZCZkKmFCHx1lZX8k0:16497:0:99999:7:::
```

Remarks. To view memory usage use the command *free –m*, the free command displays the amount of free and used memory, *-m* displays this in megabytes. If you want to repeat it each second use *watch –n 1 free –m*, the watch command executes periodically what follows, in this case *–n* means that repetition time is given in seconds. To get the running time of a program use the *time* command, e.g., *time ./test*.

Chapter 2. SYMMETRIC ENCRYPTION IN .NET

This section presents the symmetric cryptographic primitives supported by the .NET framework. All classes related to cryptography are contained within the ***System.Security.Cryptography*** namespace. The history of cryptography in Microsoft development environments starts in 1996 with the ***Win32 Cryptography API*** (Application Programming Interface) also known as ***Microsoft CryptoAPI***. Currently in .NET you will see classes that have names ending in ***CryptoServiceProvider*** and these classes are in fact wrappers over existing code from the ***Win32 Cryptography API*** (using them leads to calling code from this older API). Other class names end in ***Managed*** and these are managed code written specifically for the .NET framework. The cryptography support in .NET is mature in the sense that you have all the basic building blocks that should be needed for real-world applications. However, for more dedicated applications were you need less standard primitives or additional control over the implementation, you may want to choose a distinct environment as .NET is quite limited in this respect. Just for the sake of a rough overview, in .NET you get out-of-the-box and easy to use implementations for symmetric encryption functions (DES, 3DES, AES), hash functions (MD5, SHA1, SHA256, SHA384, SHA512, RIPEMD160), keyed hash functions (HMAC with any of the previous hash functions), public-key encryptions or signatures (RSA, DSA, EC-Diffie-Hellman-Merkle, ECDSA) and PRNGs.

2.1 SYMMETRIC ALGORITHMS, PROPERTIES AND METHODS

All of the symmetric cryptographic primitives derive from the ***SymmetricAlgorithm*** class, which is an abstract class, i.e., you cannot instantiate objects from it, rather you will work with derived concrete classes. These derived classes are: *DESCryptoServiceProvider*, *TripleDESCryptoServiceProvider*, *RC2CryptoServiceProvider*, *RijndaelManaged, AESManaged* and *AESCryptoServiceProvider*.

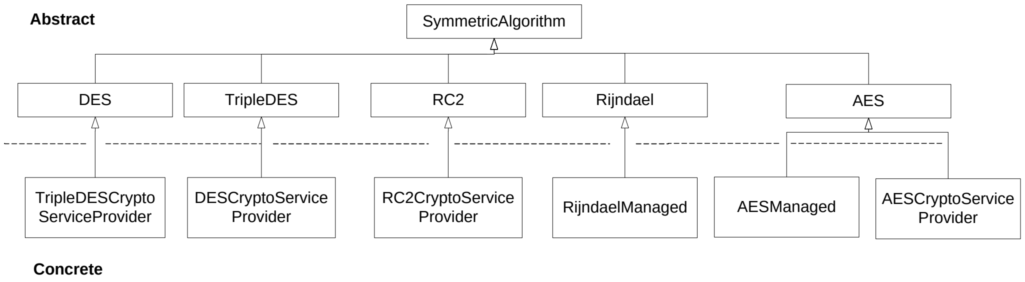


Figure 1. Symmetric encryption algorithms in .NET

Table 1 shows the properties for symmetric cryptographic algorithms in .NET. With this property list, as well as with the methods list that follows, we do not want to be exhaustive, we only try to outline what is relevant for this line of work. You must refer to MSDN for more details.

	Get/Set	***Type***	**Brief Description**
BlockSize	*g/s*	*Int*	Block size in bits
FeedbackSize	*g/s*	*Int*	Feedback size in bits (when needed, e.g., CBC, this cannot be greater than BlockSize)
IV	*g/s*	*Byte[]*	Initialization vector (IV) non-secret (must be random)
Key	*g/s*	*Byte[]*	Secret key (must be random)
KeySize	*g/s*	*Int*	Key size in bits
LegalBlockSizes	*g*	*KeySizes[]*	Block sizes in bits supported by the algorithm
LegalKeySizes	*g*	*KeySizes[]*	Key sizes in bits supported by the algorithm
Mode	*g/s*	*CipherMode*	Mode of operation (CBC is the default, the following may be supported CFB, CTS, ECB, OFB)
Padding	*g/s*	*PaddingMode*	Padding mode to fill the last block (e.g., usually none, 0xFF or zeros)

Table 1. Properties related to symmetric cryptographic algorithms in .NET

Table 2 now shows how you can assign an object that instantiates a particular symmetric implementation (DES, 3DES or Rijndael in this example) to a variable of the abstract type ***SymmetricAlgorithm***. The instantiation is done by switching over a string that contains the name of the algorithm.

```
SymmetricAlgorithm mySymmetricAlg;


public void Generate(string cipher)
{
   switch (cipher)
   {
     case "DES":
           mySymmetricAlg = DES.Create();
           break;
     case "3DES":
           mySymmetricAlg = TripleDES.Create();
           break;
     case "Rijndael":
           mySymmetricAlg = Rijndael.Create();
           break;
    }
     mySymmetricAlg.GenerateIV();
     mySymmetricAlg.GenerateKey();
}
```

Table 2. Example for instantiating an abstract object with a concrete implementation

Cryptographic streams in .NET. Before using these primitives, we have to take a brief look to another concept that is core to .NET crypto implementations: cryptographic streams. The .NET framework has a **stream-oriented design** for cryptographic primitives, an engineering idea which is beneficial because you can stream the output from one object to another and in this way the output of a crypto-stream can be directed into a file stream, memory stream, network stream, etc. Vice-versa, you can direct the output from any of the previous into a cryptographic stream. Concretely, whenever writing into a crypto-stream you will encrypt the data that is written, and vice-versa, whenever reading from the crypto stream, you will decrypt the data.

Table 3 now gives a brief overview of the methods related to symmetric cryptographic algorithms that are relevant for our scope here. Table 4 gives an example on how to encrypt an array of bytes and return the encrypted output, and similarly for decryption. The *CreateEncryptor* and *CreateDecryptor* methods return an object of type *ICryptoTransform* which can be then passed to the stream reader/writer. In Table 5 we give a more educated example that comes from the AES managed example in MSDN library (https://msdn.microsoft.com/en-us/library/system.security.cryptography.aesmanaged(v=vs.110).aspx). Note how each parameter is checked and then the ***using*** statement ensures that resources are disposed if an exception occurs (you can do the same with a ***try*** block). The using is typical for .NET style programming, so if you are keen to become an industry professional make sure to use it. Finally, the ciphertext is turned to a byte array in the following line of code: *encrypted = msEncrypt.ToArray()*.

	Return type	**Brief Description**
Clear	*void*	Zeros out all data before the object is released (relevant for security when you finished the work with the cryptographic object)
Create()	*SymmetricAlgorithm*	Creates the object
Create(String)	*SymmetricAlgorithm*	Creates the object with the string specifying the name of the particular implementation
CreateDecryptor()	*ICryptoTransform*	Creates a decryptor object
CreateDecryptor(Byte[], Byte[])	*ICryptoTransform*	Creates a decryptor object with given Key and IV
CreateEncryptor()	*ICryptoTransform*	Creates an encryptor object
CreateEncryptor(Byte[], Byte[])	*ICryptoTransform*	Creates an encryptor object with given Key and IV
Dispose()	*void*	Releases all resources used by the object
Dispose(Boolean)	*void*	Releases unmanaged and optionally managed resources used by the object
GenerateIV	*void*	Generates a random IV (note that this is already generated by CreateEncryptor and should be used only if you need a new IV)
GenerateKey	*void*	Generates a random Key (note that this is already generated by CreateEncryptor and should be used only if you need a new Key)
ValidKeySize	*bool*	Checks if a given key size is valid

Table 3. Some relevant methods for symmetric cryptographic algorithms in .NET

```
public byte[] Encrypt(byte[] mess, byte[] key, byte[] iv)
{
  mySymmetricAlg.Key = key;
  mySymmetricAlg.IV = iv;
  MemoryStream ms = new MemoryStream();
  CryptoStream cs = new CryptoStream(ms,
                       mySymmetricAlg.CreateEncryptor(),
                       CryptoStreamMode.Write);
  cs.Write(mess, 0, mess.Length);
  cs.Close();
  return ms.ToArray();
}

public byte[] Decrypt(byte[] mess, byte[] key, byte[] iv)
{
  byte[] plaintext = new byte[mess.Length];
  mySymmetricAlg.Key = key;
  mySymmetricAlg.IV = iv;
  MemoryStream ms = new MemoryStream(mess);
  CryptoStream cs = new CryptoStream(ms,
                       mySymmetricAlg.CreateDecryptor(),
                       CryptoStreamMode.Read);
  cs.Read(plaintext, 0, mess.Length);
  cs.Close();
  return plaintext;
}
```

Table 4. A rather quick way for building encryption and decryption functions

Note: example reproduced from MSDN library (https://msdn.microsoft.com/en-us/library/system.security.cryptography.aesmanaged(v=vs.110).aspx)

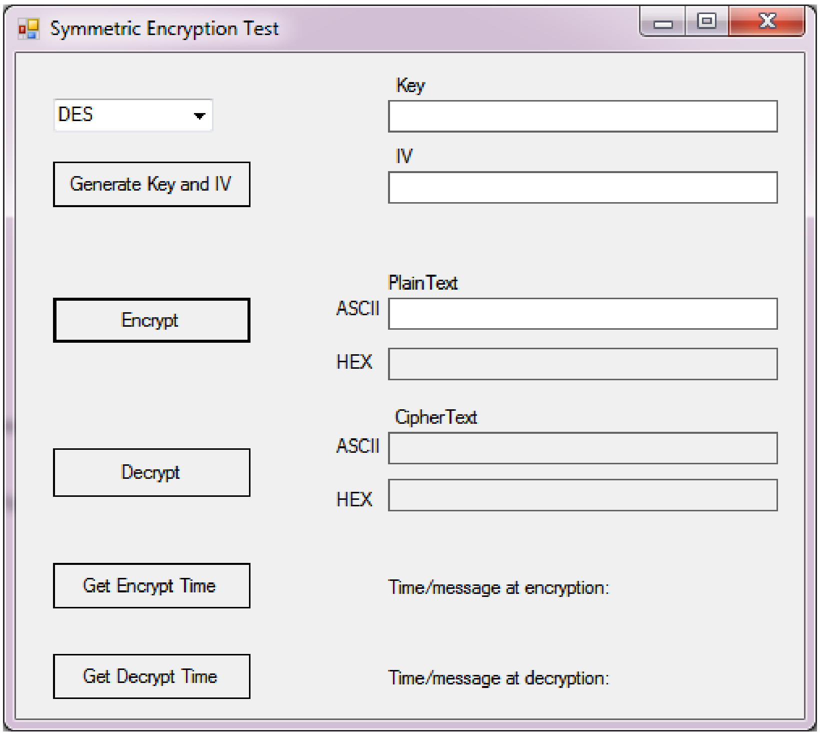
```
// Check arguments.
if (plainText == null || plainText.Length <= 0)
                throw new ArgumentNullException("plainText");
if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
if (IV == null || IV.Length <= 0)
```

```
                throw new ArgumentNullException("Key");
byte[] encrypted;
// Create an AesManaged object
// with the specified key and IV.
using (AesManaged aesAlg = new AesManaged()){

      aesAlg.Key = Key;
      aesAlg.IV = IV;

      // Create a decrytor to perform the stream transform.
      ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
aesAlg.IV);

      // Create the streams used for encryption.
      using (MemoryStream msEncrypt = new MemoryStream())
      {
            using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                    encryptor, CryptoStreamMode.Write))
            {
                  using (StreamWriter swEncrypt = new
StreamWriter(csEncrypt))
                  {

                      //Write all data to the stream.
                      swEncrypt.Write(plainText);
                  }
                  encrypted = msEncrypt.ToArray();
            }
     }
}
// Return the encrypted bytes from the memory stream.
return encrypted;
```

Table 5. A more educated example from Microsoft's MSDN library (note how the arguments are checked and the *using* directive)

2.2 EXERCISES

1. Write a C# application that allows a user to select an encryption algorithm from a Combo Box, generate keys, encrypt and decrypt messages. Display the plain text and cipher text both in ASCII and HEX and similarly the Keys and IVs; also display the time required by the encryption and decryption operations. A suggested interface is below, but feel free to modify it at will.



2. You are required to evaluate the computational costs of symmetric cryptographic primitives in .NET. Results have to be presented in a tabular form as shown below and measured in seconds/block then bytes/second considering both streams from memory and from the local hard-drive.

	AES (CSP) (128 bit)	AES (CSP) (256 bit)	AES (Managed) (128 bit)	AES (Managed) (256 bit)	Rijndael (Managed) (128 bit)	Rijndael (Managed) (256 bit)	DES (CSP) (56 bit)	3DES (CSP) (168 bit)
seconds/block								
bytes/second (from RAM)								
bytes/second (from HDD)								

Table 6. Computational cost for symmetric cryptographic primitives

3. Exhaustive search for the key. You are required to adapt the code from Section 1 for cracking passwords (feel free to write your own code if you want) in order to break the following DES ciphertext knowing that the plaintext starts with the 'asdf' letters and the key has the last 6 bytes set to 0 (that is, you have to perform an exhaustive search over the first 2 bytes). By breaking the ciphertext, we understand here finding the encryption key and the message.

IV in Hex: 01092C61619EE95E

Ciphertext in Hex:
CD56D268F00D5CABE4A649A3028F4EC34BA8C23CA26ADD8A5BBAE934C8B286DF

Remarks. For Exercise 1 you can start by recycling some of the code below.

```
using System.Security.Cryptography;
using System.IO;

namespace Example
{
```

```
    public partial class SymEnc : Form
    {

        ConversionHandler myConverter = new ConversionHandler();

        SymmetricAlgorithm mySymmetricAlg;


        public SymEnc()
        {
            InitializeComponent();
        }

        public void Generate(string cipher)
        {
            switch (cipher)
            {
                case "DES":
                    mySymmetricAlg = DES.Create();
                    break;
                case "3DES":
                    mySymmetricAlg = TripleDES.Create();
                    break;
                case "Rijndael":
                    mySymmetricAlg = Rijndael.Create();
                    break;
            }
            mySymmetricAlg.GenerateIV();
            mySymmetricAlg.GenerateKey();
        }

        public byte[] Encrypt(byte[] mess, byte[] key, byte[] iv)
        {
            mySymmetricAlg.Key = key;
            mySymmetricAlg.IV = iv;
            MemoryStream ms = new MemoryStream();
            CryptoStream cs = new CryptoStream(ms,
                                                     mySymmetricAlg.Cre
                                                     ateEncryptor(),
                                                     CryptoStreamMode.W
                                                     rite);
            cs.Write(mess, 0, mess.Length);
            cs.Close();
            return ms.ToArray();
        }

        public byte[] Decrypt(byte[] mess, byte[] key, byte[] iv)
        {
            byte[] plaintext = new byte[mess.Length];
            mySymmetricAlg.Key = key;
```

```
            mySymmetricAlg.IV = iv;
            MemoryStream ms = new MemoryStream(mess);
            CryptoStream cs = new CryptoStream(ms,
                                  mySymmetricAlg.CreateDecryptor(),
                                  CryptoStreamMode.Read);
            cs.Read(plaintext, 0, mess.Length);
            cs.Close();
            return plaintext;
        }

        private void buttonEnc_Click(object sender, EventArgs e)
        {
            byte[] ciphertext =
            Encrypt(myConverter.StringToByteArray(textBoxPlain.Text),
            myConverter.HexStringToByteArray(textBoxKey.Text),myConvert
            er.HexStringToByteArray(textBoxIV.Text));
            textBoxCipher.Text =
myConverter.ByteArrayToString(ciphertext);
            textBoxCipherHex.Text =
myConverter.ByteArrayToHexString(ciphertext);
            textBoxPlainHex.Text =
            myConverter.ByteArrayToHexString(myConverter.StringToByteAr
            ray(textBoxPlain.Text));
        }

        private void buttonDec_Click(object sender, EventArgs e)
        {
             byte[] plaintext  =
             Decrypt(myConverter.HexStringToByteArray(textBoxCipherHex.
             Text),

            myConverter.HexStringToByteArray(textBoxKey.Text),myConvert
            er.HexStringToByteArray(textBoxIV.Text));
            textBoxPlain.Text =
myConverter.ByteArrayToString(plaintext);
            textBoxPlainHex.Text =
myConverter.ByteArrayToHexString(plaintext);
        }

        private void buttonGen_Click(object sender, EventArgs e)
        {
            Generate(comboBoxCipher.Text);
            textBoxKey.Text =
myConverter.ByteArrayToHexString(mySymmetricAlg.Key);
            textBoxIV.Text =
myConverter.ByteArrayToHexString(mySymmetricAlg.IV);
        }

        private void buttonEncTime_Click(object sender, EventArgs e)
        {
```

```
            mySymmetricAlg.GenerateIV(); // generates a fresh IV
            mySymmetricAlg.GenerateKey(); // generates a fresh Key

            MemoryStream ms = new MemoryStream();
            CryptoStream cs = new CryptoStream(ms,
                                 mySymmetricAlg.CreateEncryptor(),
                                 CryptoStreamMode.Write);
            byte[] mes_block = new byte[8];
            long start_time = DateTime.Now.Ticks;
            int count = 10000000;
            for (int i = 0; i < count; i++)
            {
                cs.Write(mes_block, 0, mes_block.Length);
            }
            cs.Close();
            double operation_time = (DateTime.Now.Ticks - start_time);
            operation_time = operation_time / (10*count); // 1 tick is
                                                             100 ns,
                                                             i.e., 1/10
                                                             of 1 us
            labelEncTime.Text = "Time for encryption of a message
                                block: " + operation_time.ToString() +
                                " us";
        }

    }
}
```

```
    class ConversionHandler
    {

       public byte[] StringToByteArray(string s)
        {
            return CharArrayToByteArray(s.ToCharArray());
        }

        public byte[] CharArrayToByteArray(char[] array)
        {
            return Encoding.ASCII.GetBytes(array, 0, array.Length);
        }

        public string ByteArrayToString(byte[] array)
        {
            return Encoding.ASCII.GetString(array);
```

```
        }

        public string ByteArrayToHexString(byte[] array)
        {
            string s = "";
            int i;
            for (i = 0; i < array.Length; i++)
            {
                s = s + NibbleToHexString((byte)((array[i] >> 4) &
                        0x0F)) + NibbleToHexString((byte)(array[i] &
                        0x0F));
            }
            return s;
        }

        public byte[] HexStringToByteArray(string s)
        {
            byte[] array = new byte[s.Length / 2];
            char[] chararray = s.ToCharArray();
            int i;
            for (i = 0; i < s.Length / 2; i++)
            {
                array[i] = (byte)(((HexCharToNibble(chararray[2 * i])
                          << 4) & 0xF0) | ((HexCharToNibble(chararray[2
                          * i + 1]) & 0x0F)));

            }
            return array;
        }

        public string NibbleToHexString(byte nib)
        {
            string s;
            if (nib < 10)
            {
                s = nib.ToString();
            }
            else
            {
                char c = (char)(nib + 55);
                s = c.ToString();
            }
            return s;
        }

        public byte HexCharToNibble(char c)
        {
            byte value = (byte)c;
            if (value < 65)
            {
```

```
                value = (byte)(value - 48);
            }
            else
            {
                value = (byte)(value - 55);
            }
            return value;
        }
    }
```

Chapter 3. HASH FUNCTIONS AND MAC CODES IN .NET

This section presents the hash functions and their immediate derivative *Message Authentication Codes (MAC)* that are supported by the .NET framework. We also make use of random number generators.

The .NET framework supports the now deprecated but still largely used MD5 (128 bit) and SHA1 (160 bit). Besides these, there is also support for the (soon to be replaced) current standard SHA2 in all three output sizes 256, 384 and 512 bit and the less frequent RIPEMD (160 bit).

MACs (Message Authentication Codes) are also named keyed hash functions since they are built from a hash function with the use of a secret key. But there are also exceptions to this rule and it happens for MAC codes to be built from symmetric encryption functions rather than hash functions. The .NET framework contains one such exception which is the *MACTripleDES*, a MAC code build on 3DES. The other MAC code that is supported by .NET is HMAC, which is indeed a keyed hash function and the preferred alternative, it can be built on any of the hash functions available in the framework: MD5, SHA1, SHA2 or RIPEMD. Figure 1 shows the class organization for hash algorithms and MAC codes, we can see again the distinction between abstract and concrete classes.

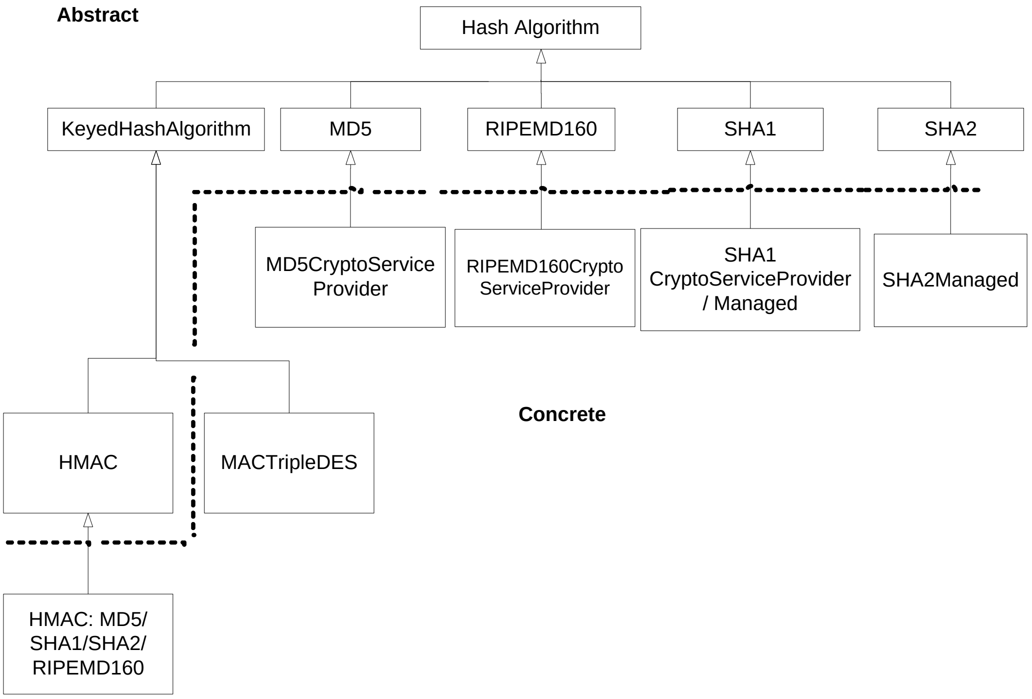


Figure 1. Hash functions and keyed hash functions in .NET

3.1 HASH FUNCTIONS

Hash functions are derived from the abstract class *HashAlgorithm* located in the aforementioned *System.Security.Cryptography* namespace. Some properties and methods are outlined in Tables 1 and 2.

	Get/Set	***Type***	**Brief Description**
InputBlockSize	*g*	*Int*	Bit size of the input block, returns 1 unless overwritten
OutputBlockSize	*g*	*Int*	Bit size of the output block, returns 1 unless overwritten
HashSize	*g*	*Int*	Bit size of the hash
Hash	*g*	*Byte[]*	Value of the hash

Table 1. Some properties for hash functions in .NET

	Return type	**Brief Description**
Create()	*HashAlgorithm*	Creates the object (SHA1 is the default instance)
Create(String)	*HashAlgorithm*	Creates the object with the string specifying the name of the particular implementation given as string (MD5, SHA1, etc.)
ComputeHash(Byte[])	*Byte[]*	Computes the hash from a byte array
ComputeHash(Stream)	*Byte[]*	Computes the hash from a stream object
ComputeHash(Byte[], Int32, Int32)	*Byte[]*	Computes the hash from a specific region of a byte array
TransformBlock(byte[] inputBuffer, int inputOffset, int inputCount, byte[] outputBuffer, int outputOffset)	*Int*	Computes the hash of a specified region of a byte array and copies the region to the specified region of the output byte array. Return the number of bytes written.
TransformFinalBlock(byte[] inputBuffer, int inputOffset, int inputCount)	*Byte[]*	Computes the hash of a specified region of a byte array, returns a copy a of the part of the input that is hashed

Table 2. Some methods for hash functions in .NET

To compute the hash of a byte array or stream you can simply call the *ComputeHash* method as outlined in Table 3. In this example we also used a *RandomNumberGenerator* object to generate some arbitrary values that are later hashed. To generate random values, you simply have to create a *RandomNumberGenerator* object and make a call to the *GetBytes* method on a specific byte array.

```
MD5CryptoServiceProvider myMD5 = new MD5CryptoServiceProvider();
RandomNumberGenerator rnd = RandomNumberGenerator.Create();
byte[] input = new byte[20];
byte[] hashValue;
//generates some random input
rnd.GetBytes(input);
//computes the hash
hashValue = myMD5.ComputeHash(input);
```

Table 3. Example for generating some random bytes and computing their hash

In Table 4 we then show how to compute the hash of a given file, this is a frequently used procedure to check the integrity of files, or to compare if two files (or objects) are the same, as only identical objects can hash to the same value (assuming the hash function is collision free).

```
FileStream fileStream = new FileStream("C:\\TEMP\\x.pdf",
FileMode.Open);
fileStream.Position = 0;
hashValue = myMD5.ComputeHash(fileStream);
```

Table 4. Example for computing the hash of a given file

3.2 KEYED HASH FUNCTIONS

The .NET framework provides an implementation for the HMAC keyed hash function. The properties and methods for *KeyedHashAlgorithm* objects are almost identical to that of *HashAlgorithm* (a class which they do inherit). The only additional property, is the one to get or set the key as outlined in Table 5.

	Get/Set	***Type***	**Brief Description**
Key	*g/s*	*Byte[]*	Value of the key for the HMAC

Table 5. The *Key* property of keyed hash algorithms in .NET

In Tables 6 and 7 we show how to instantiate a HMAC with a particular hash function, how to generate the authentication tag with *ComputeMAC(byte[] mes, byte[] key)* and then verify it with *CheckAuthenticity(byte[] mes, byte[] mac, byte[] key)*.

```
private HMAC myMAC;

public MACHandler(string name)
{
if (name.CompareTo("SHA1") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA1(); }
if (name.CompareTo("MD5") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACM
                                     D5(); }
if (name.CompareTo("RIPEMD") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACR
                                     IPEMD160(); }
if (name.CompareTo("SHA256") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA256(); }
if (name.CompareTo("SHA384") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA384(); }
if (name.CompareTo("SHA512") == 0) { myMAC = new
                                     System.Security.Cryptography.HMACS
                                     HA512(); }
}
```

Table 6. Creating a HMAC object with a particular hash function

```
public bool CheckAuthenticity(byte[] mes, byte[] mac, byte[] key)
{
  myMAC.Key = key;
  if (CompareByteArrays(myMAC.ComputeHash(mes), mac, myMAC.HashSize /
8) == true)
   {
      return true;
   }
  else
   {
```

```
      return false;
   }
}

public byte[] ComputeMAC(byte[] mes, byte[] key)
{
  myMAC.Key = key;
  return myMAC.ComputeHash(mes);
}

public int MACByteLength()
{
  return myMAC.HashSize / 8;
}

private bool CompareByteArrays(byte[] a, byte[] b, int len)
{
  for (int i = 0; i < len; i++)
    if (a[i] != b[i]) return false;
  return true;
}
```

Table 7. Computing the HMAC and then verifying the authenticity of a message

3.3 HASH FUNCTIONS AND MAC CODES AS CRYPTOSTREAMS

A final trick that may be useful to know is that you can pass hash functions or HMACs as transformations embedded into *CryptoStreams*. In Table 8 we show such an example. The streams that we use will not store the data that is written into them, i.e., they receive Stream.Null at initialization. In order to retrieve the hash or HMAC value we then simply call the Hash property of the cryptographic objects, i.e., *hmac.Hash* and *hash.Hash*.

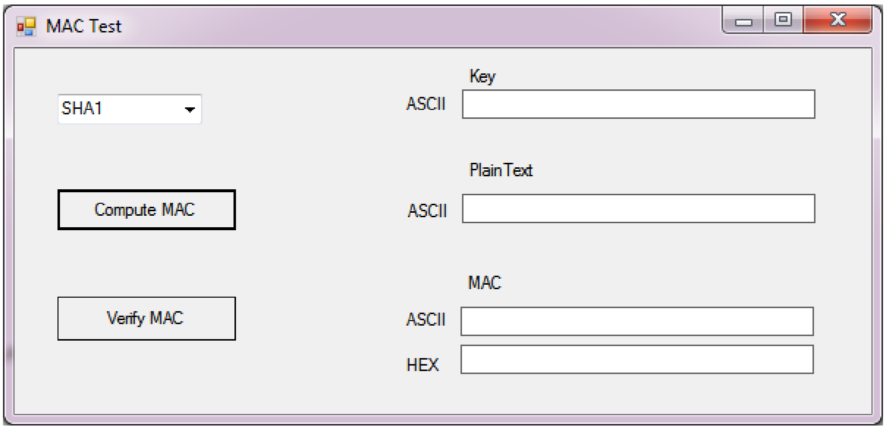
```
RandomNumberGenerator rnd = RandomNumberGenerator.Create();
byte[] key = new byte[16];
rnd.GetBytes(key);
byte[] input = new byte[20];
rnd.GetBytes(input);

HMACSHA256 hmac = new HMACSHA256(key);
SHA256Managed hash = new SHA256Managed();
```

```
CryptoStream cs_hmac = new CryptoStream(Stream.Null, hmac,
CryptoStreamMode.Write);
CryptoStream cs_hash = new CryptoStream(Stream.Null, hash,
CryptoStreamMode.Write);

cs_hmac.Write(input, 0, input.Length);
cs_hmac.Close();

cs_hash.Write(input, 0, input.Length);
cs_hash.Close();
```

Table 8. Example for HMACSHA256 and SHA256 used in *CryptoStreams*

3.4 EXERCISES

2. Write a C# application that allows a user to select a Hash or HMAC algorithm from a Combo Box, generate keys (in case of HMAC), hash messages and verify (in case of HMAC) their hashes. Display the plain text and hash both in ASCII and HEX; also display the time required by the hash and HMAC operations. A suggestion for starting the interface is below, but feel free to modify it at will. Results should be presented in a tabular form as shown below.



	SHA1 (CSP)	SHA1 (Managed)	SHA256 (CSP)	SHA256 (Managed)	SHA384 (CSP)	SHA256 (Managed)	SHA512 (CSP)	SHA512 (Managed)	MD5 (CSP)	RIPEMD (Managed)
seconds/block										
bytes/second (from RAM)										
bytes/second (from HDD)										

Table 9. Computational cost for hash functions

2. Write a program that searches, by generating random values, for hashes that have all the last k bits set to 0 (k is given as parameter by the user). Give an estimation to find such values for a given k.

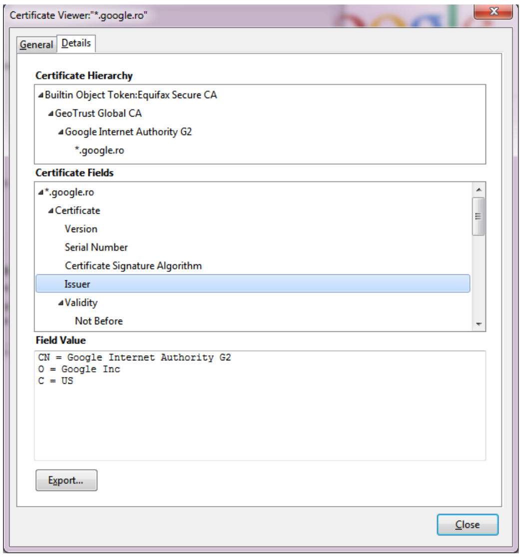
Remark. You can recycle some of the code below for the interface of exercise 1.

```
private void buttonCompute_Click(object sender, EventArgs e)
{
  MACHandler mh = new MACHandler(comboBoxMAC.Text);
  byte[] mac =
              mh.ComputeMAC(myConverter.StringToByteArray(textBoxPlain.
              Text),myConverter.StringToByteArray(textBoxKey.Text));
  textBoxMAC.Text = myConverter.ByteArrayToString(mac);
  textBoxMACHEX.Text = myConverter.ByteArrayToHexString(mac);
}


private void buttonVerify_Click(object sender, EventArgs e)
{
   MACHandler mh = new MACHandler(comboBoxMAC.Text);
```

```
   if
              (mh.CheckAuthenticity(myConverter.StringToByteArray(textB
              oxPlain.Text),
              myConverter.HexStringToByteArray(textBoxMACHEX.Text),myCo
              nverter.StringToByteArray(textBoxKey.Text)) == true)
   {
      System.Windows.Forms.MessageBox.Show("MAC OK !!!");
   }
   else
   {
      System.Windows.Forms.MessageBox.Show("MAC NOT OK !!!");
   }
}
```

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.



Certificate Hierarchy

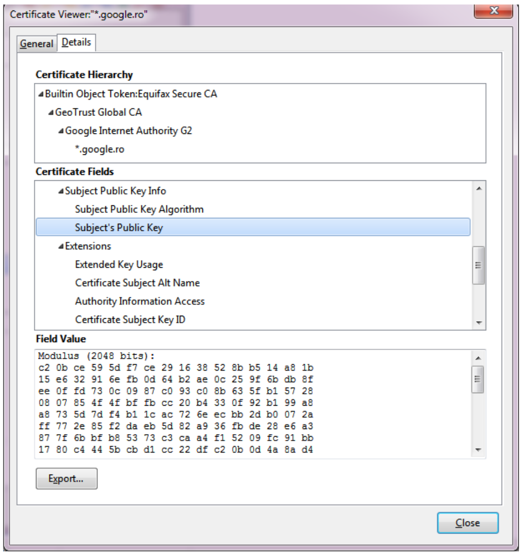
- Builtin Object Token:Equifax Secure CA
  - GeoTrust Global CA
    - Google Internet Authority G2
      - *.google.ro

Certificate Fields

- *.google.ro
  - Certificate
    - Version
    - Serial Number
    - Certificate Signature Algorithm
    - Issuer
    - Validity
      - Not Before

Field Value

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



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

```
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 straightforward 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.

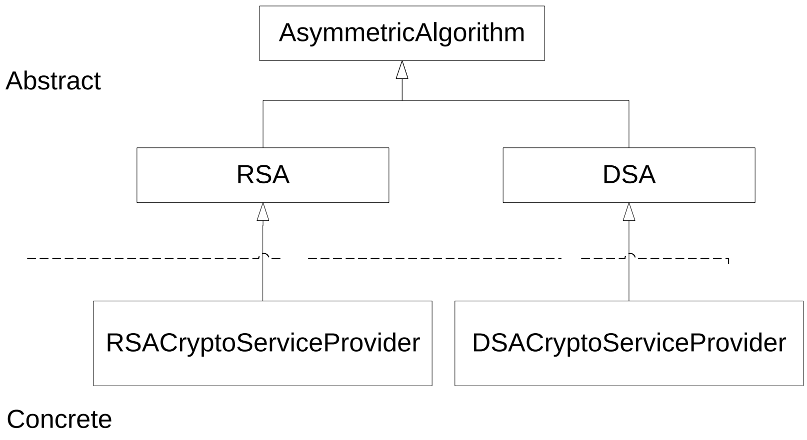


Figure 2. The RSA and DSA clases in .NET

	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., $q^{-1}\ \mathrm{mod p}$,
- ✓ **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>
```

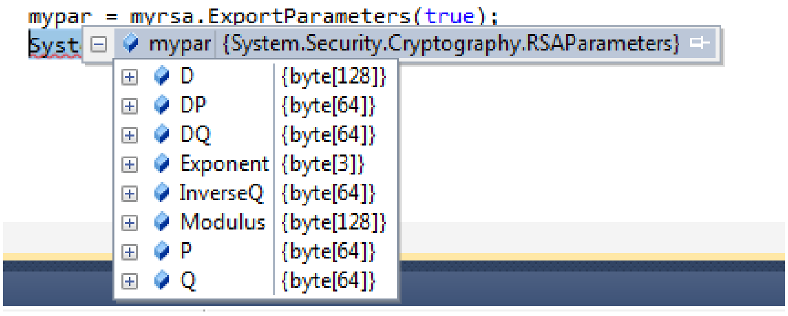
Table 5. RSA key exported as XML string with private parameters

```
<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;
m[1]=94;
m[2]=85;
m[3]=235;
	d[0]=16;
	d[1]=158;
	d[2]=240;
	d[3]=222;
		m[0]=184;
		m[1]=180;
		m[2]=103;
		m[3]=69;
			dp[0]=66;
			dp[1]=59;
			dp[2]=152;
			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=
```

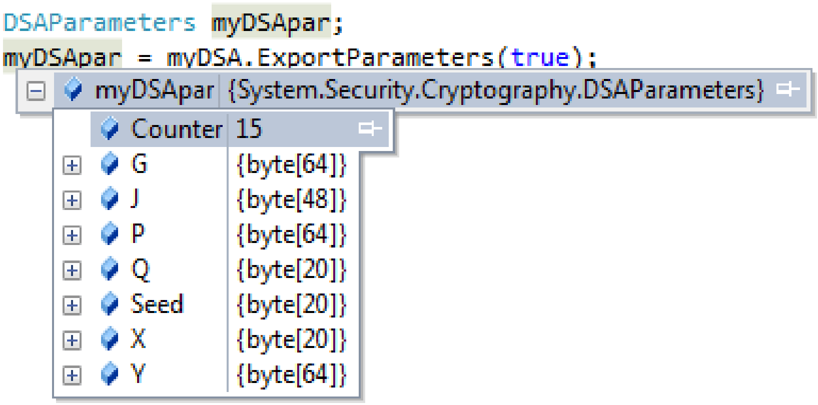
```
</J>
<Seed>
nVHH51WY6AQqRGBXYDg+zQnHF5s=
</Seed>
<PgenCounter>
Ag==
</PgenCounter>
<X>
NhAM2W6d+VISPUZ967Zfc8v8D+8=
</X>
</DSAKeyValue>
```

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

Chapter 6. COMPUTATIONAL PROBLEMS BEHIND PUBLIC-KEY CRYPTOSYSTEMS, BIGINTEGERS IN JAVA

In this section we pay attention to computational problems that stay at the core of public key cryptosystems, RSA in particular. We exemplify computational problems with the help of the *BigInteger* class from Java. Rather than briefing through the capabilities of this class, we take a problem based approach in which we try to underline the math behind cryptosystems such as the RSA (pointing on issues that potentially cause insecurity). A shortcoming of this section is that we do not describe the particular algorithms behind these computations, however some of the algorithms are described during the lectures and here we try to fix the notions by playing with numbers.

6.1 THE JAVA BIGINTEGER CLASS

The Java *BigInteger* class allows working with arbitrary precision integers. There is virtually no limit on their size, except for the memory available. However, in public key cryptosystems we usually work with integers that are in the order of several thousands of bits, e.g., 1024-4096 in case of the RSA, so you should imagine this as the practical size that we target. To initialize a *BigInteger* is fairly simple, the constructor of the class can also take strings, for example,

BigInteger two = new BigInteger("2");

creates a *BigInteger* with value 2. You can initialize the integer with a value of your choice, e.g.,

BigInteger exponent = new BigInteger("65537");

Then operations are simply performed by calling the related methods. For example if you want to compute an exponentiation $2^{65537} \bmod 3$ simply call:

BigInteger result = two.modPow(exponent, new BigInteger("3"));

In Table 1 we summarize the arithmetic operations and the equivalent Java BigInteger's methods.

Arithmetic Operation	**Java BigInteger Method**
additions and subtractions (+, -)	subtract(BigInteger val) add(BigInteger val)
multiplications and divisions (*, /),	multiply(BigInteger val) divide(BigInteger val) divideAndRemainder(BigInteger val) mod(BigInteger m) remainder(BigInteger val)
comparisons (<, >)	compareTo(BigInteger val) max(BigInteger val) min(BigInteger val)
exponentiation and modular exponentiation, a^x	modPow(BigInteger exponent, BigInteger m) pow(int exponent)
greatest common divisor (GCD) and multiplicative inverse, i.e., x^{-1}	gcd(BigInteger val) modInverse(BigInteger m)
primality testing	isProbablePrime(int certainty) probablePrime(int bitLength, Random rnd)

Table 1. A summary of arithmetic operations and the corresponding methods in Java

6.2 SOLVED EXERCISES

The private exponent reveals the factorization of the modulus. This is a commonly known property of the RSA. It is also the reason for which a modulus cannot be shared by two distinct entities even if they use distinct public exponents (since the private exponent of each of them can be used to factor the modulus and recover the private exponent of the other). This problem is also referred as the common modulus problem.

Let the following RSA key:

$$K_1 : \{n = 837210799, e = 7, d = 478341751\}$$

Show how the modulus can be factored given the private key and find the private exponent for the following key:

$$K_3 : \{n = 837210799, e = 17, d = ?\}.$$

Solution 1. *The mathematical relation between the private and public RSA exponents is the following:*

$$d \cdot e \equiv 1 \bmod \phi(n)$$

This implies that there exists a number k such that

$$d \cdot e = 1 + k \cdot \phi(n).$$

Since

$$\phi(n) = (p-1)(q-1) = p \cdot q - p - q + 1$$

It follows that

$$d \cdot e = 1 + k \cdot (p \cdot q - p - q + 1)$$

Rearranging the terms we get

$$pq + 1 - \frac{d \cdot e - 1}{k} = p + q.$$

We know all values from the left side, except for k. *However, by closely examining the previous relation* $d \cdot e = 1 + k \cdot (p \cdot q - p - q + 1)$ *since on the right side* $p \cdot q$ *is much larger than* $-p - q + 1$ *we are not far by approximating k as:*

$$k \approx \left\lceil \frac{d \cdot e - 1}{p \cdot q} \right\rceil = \left\lceil \frac{d \cdot e - 1}{n} \right\rceil$$

In our case, starting from the already known key we get:

$$k \approx \left\lceil \frac{7 \cdot 478341751 - 1}{837210799} \right\rceil = 4$$

It follows that:

$$p + q = \frac{4 \cdot (837210799 + 1) + 1 - 7 * 478341751}{4} = 112736$$

This implies that p and q can be extracted as roots of the equation $x^2 - Sx + P = 0$ *where* $S = 112736$ *and* $P = 837210799$. *By elementary calculations, we get:*

$$\Delta = 112736^2 - 4 \cdot 837210799 = 9360562500$$

The roots follow as:

$$x_1 = \frac{112736 + \sqrt{9360562500}}{2} = 104743 \quad \textit{and}$$

$$x_2 = \frac{112736 - \sqrt{9360562500}}{2} = 7993$$

These are the factors of the modulus. Finding the second private exponent is now trivial as:

$$d = e^{-1} \bmod (p-1)(q-1) \Rightarrow d = 17^{-1} \bmod 837098064 = 246205313$$

Solution 2. *The private exponent always decrypts a message encrypted with the public one, since:*

$$\forall x \in Z_n, x = \left(x^e\right)^d \bmod n$$

Given the values from the first key we always have:

$$x = \left(x^7\right)^{478341751} \bmod 837210799$$

By multiplying with x^{-1} *and rearranging we get:*

$$x^{7\cdot 478341751-1} = 1 \bmod 837210799$$

Dividing by 2 we get:

$$\left(x^{\frac{7\cdot 478341751-1}{2}}\right)^2 = 1 \bmod 837210799$$

This means that the right quantity is a square root of 1. To eliminate the two trivial roots of 1, i.e., +1 and -1, we continuously divide the exponent until we get a non-trivial root. For example, let us fix $x = 10$ *and compute:*

$$x^{\frac{7\cdot 478341751-1}{2}} = 1, x^{\frac{7\cdot 478341751-1}{4}} = 1, x^{\frac{7\cdot 478341751-1}{8}} = 1, x^{\frac{7\cdot 478341751-1}{16}} = 562155682$$

It is easy to note that when dividing the exponent with 16 the result is no longer 1. For this final result we have:

$$cmmdc\left(562155682 - 1, n\right) = 7993 = p$$
$$cmmdc\left(562155682 + 1, n\right) = 104743 = q$$

In this way we have successfully extracted the factors of n*. The mathematical explanation is that we have:*

$$\left(x^{\frac{7\cdot 478341751-1}{16}}\right)^2 \equiv 1 \bmod n \Leftrightarrow \left(x^{\frac{7\cdot 478341751-1}{16}} - 1\right)\left(x^{\frac{7\cdot 478341751-1}{16}} + 1\right) \equiv 0 \bmod n$$

and since $x^{\frac{7\cdot 478341751-1}{16}} \neq \pm 1$ *it means that the two factors contain the prime numbers that divide* n.

Small encryption exponents. While small exponents are preferred for encryption because they result in faster operation, small exponents are known to cause insecurity. The .NET framework has the default exponent set to 65537, this should be secure, but it may be tempting to use even smaller exponents. Consider the following two 1536 and 2048 bit modules taken from the RSA challenge website http://www.rsasecurity.com/rsalabs/node.asp?id=2093

n_1=

1847699703211741474306835620200164403018549338663410171471785774910651696711161249859337684305435744585616061544571794052229717732524660960646946071249623720442022269756756687378427562389508764678440933285157496578843415088475528298186726451339863364931908084671990431874381283363502795470282653297802934916155811881049844908319545009848393775227257052578591944993870073695755688436933812779613089230392569695253261620823676490316036551371447913932347169566988069

n_2=

25195908475657893494027183240048398571429282126204032027777137836043662020707595556264018525880784406918290641249515082189298559149176184502808489120072844992687392807287776735971418347270261896375014971824691165077613379859095700097330459748808428401797429100642458691817195118746121515172654632282216869987549182422433637259085141865462043576798423387184774447920739934236584823824281198163815010674810451660377306056201619676256133844143603833904414952634432190114657544454178424020924616515723350778707749817125772467962926386356373289912154831438167899885040445364023527381951378636564391212010397122822120720357

By this exercise we show that even if the factorization of these numbers is unknown (these challenge numbers were not yet factored, so it is impossible for us to know their factorization), one can still recover encrypted values in certain situations if the exponents are small. Consider that one fixes an encryption exponent $e = 2$ (this is in fact known as the Rabin cryptosystem and is a secure cryptosystem when correctly used, see the lecture material for more details) and that one encrypts the same message m once with each modulus, i.e., $c_1 = m^2 \mod n_1$, $c_2 = m^2 \mod n_2$. Given the result of the encryptions below, you are requested to find the encrypted message:

c_1=

1720824975522517857539467309146518060382842270514896093391979291030656292239
7291446654035136859446266905140522147597644944431643498057575862023479413245
6638260412096493538625812249998880361757163409597018001190001744747405240965
7500820140866171389821089899978493473235156488326073675749875367732149010528
9244104109064444335973488450882364503785143338799248614163518428477608940469
9678849571206887860878689927075639507531091535187214291140378602914898718344
7449947

c_2=
4561642280956381246774642331705575104523442518306294887033201504008906454855
8878555145972657908956759775539747979197737797768926554418702738975251318948
7102258520443358104409325508073221395545765319081041834133569912754811011387
3635190699932165850542152382657518899992710162713201334532551245793969597202
6692191157400036070478620074907493119547542465852819192370184492356694178657
6698578327560649299302223024036233077234207232288187628580786589383228234629
4300028016342171410187938861009812975635715641457865781951720724292241356964
6111551957961184286656146057704287329146644239215935313741848147782402529568
44983980

Solution. *The mistake comes from the fact that the small encryption exponent allows one to recover the message by squaring the output composed via the Chinese Remaindering Theorem (CRT). We show how this can be done in what follows. CRT implies that the following result holds:*

Iff $\begin{cases} m^2 \equiv c_1 \mod n_1 \\ m^2 \equiv c_2 \mod n_2 \end{cases}$ *then there exists a unique m modulo* $n_1 \cdot n_2$.

But message `m` *was encrypted with the first modululs, this means it cannot exceed 1536 de bits. Therefore the square of the message has at most 2x1536 = 3072 bits. CRT allows one to retrieve a solution modulo* $n_1 \cdot n_2$*, n.b., moreover, this solution is unique. Since the two modules have 1536 and 2058 bits respectively it means that this solution is unique for up to 1536+2048 = 3584 bits and thus message* `m` *can be fully recovered as square root of the value retrieved via CRT. We show how this can be done by using the CRT solution offered by Gauss. First, we compute the modular inverses:*

$n_1^{-1} \bmod n_2 =$

14310987589421656595052001273606996601993205437791295923061219355358489326217220479964791723952051824847099259813450868235923610986761167139297230637140711591789321531779860915129975265082841364126014370329155407443919355323334425193123995577457586594368899226059650898095208343254419028472810136331854686339459392688075632057376317621886415267112093032817075738101542972428151967224553698934704282194670757975832865425047290849934241209824297863258898100147349976522660848513214782258806066209377656764702897124115159948757949075408546009463695334587761301941756044850637877986046178457086103042865469965847913776536

$n_2^{-1} \bmod n_1 =$

79822742293307335384489483161431538390245026454391226276909057792674233805796669352381282742024598053601691704533999669231177701817255927560148204696029659137909256560710045948920441282490872334259240595183320111854009654964710771311177195733351955713468728607066480923161798282214922420839905945842601231654697317438769934003745932335839323093556548609036647293884293624133796731609301787916832516880666680181005046190919436075737355630558837491016361377472345O

Then we use Gauss's solution for the CRT and compute:

$m^2 = \left(c_1 n_2 n_2^{-1} \bmod n_1 + c_2 n_1 n_1^{-1} \bmod n_2\right) \bmod n_1 n_2 =$

4024840927937178156259470331471591003484786922536638102254069791417585602944732917136098048248669251363580202404678911735557994243268711309574801864624906335017940237868171255569401324570900934477886232467631201564000784516895533937832227067091147558600244462890133397778221666551093553199704408884732857724216266011039547799164354879332317670210865470985542398624300873931777616205464930931536998083441900345650126568812496811237279343495905746190180574213036865242679883549911146730234579613576919248080423603916547270288585014116973253480963657921947810342590414657027258881503711927348350396595815197081264282043765932748890450601215737135269682583819938149430504606616277189 2

75083123873943549970558612016817243280483663927948950510612617417357
297285884203981954351050666622817897351291284744036

Now we can extract the encrypted message as the square root of the previous message, i.e.,:

$m = \sqrt{m^2} =$
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
200620062006200620062006200620062006200620062006200620062006

Balanced vs. unbalanced RSA. The RSA version in which the two factors p and q have the same size is also referred as balanced RSA. An unbalanced version of the RSA was also proposed, it benefits from a large modulus (harder to factor, thus increased security) but still fast for decryption if this is performed via the smaller factor. Unbalanced RSA assumes the use of a small p (e.g., several hundred bits) and a larger q (e.g., several thousand bits). Only messages smaller than p are encrypted and then decryption is performed modulo p (this can be done only by the owner of the private key who is in possession of p). For correct encryption, a bound l on the size of the plaintext is made public (this does not make the scheme unsafe, it is simply the bitlength of p which does not make factorization trivial). We give a small numerical example:

Key generation:

$$p = 541, q = 104729, e = 7, l = 200$$
$$\Rightarrow n = 56658389, \phi(n) = (p-1)(q-1) = 56553120,$$
$$d = e^{-1} \bmod (p-1) = 463$$

Encryption:

$$m = 300 \Rightarrow c = 300^7 \bmod 56658389 = 18157376$$

Decryption:

$$m = 18157376^{463} \bmod 541 = 300$$

Show how a CCA2 (Chosen Ciphertext Attack) attack can be mounted such that the adversary can recover the private key. Use the previous numbers to illustrate the attack.

Solution. *The CCA2 attack assume that the adversary has unlimited access to the decryption machine, i.e., the machine accepts to decrypt messages at his choice. The adversary can cheat and encrypt a message that is larger than the bound* l *, e.g.,*

$$c = 1000^{7} \bmod 56658389 = 27641532$$

The decryption machine performs decryption according to the rules and answers with:

$$m = 27641532^{463} \bmod 541 = 459$$

Now the adversary can use this response to factor the modulus as:

$$\gcd(1000 - 459, n) = 541$$

Thus, the adversary can factor the modulus and completely break the cryptosystem. The mathematical fact behind this attack is trivial. Since $x \equiv \left(x^{e}\right)^{d} \bmod p$ *but* $x \neq \left(x^{e}\right)^{d}$ *mod p (note that* $\gg$ *p) it follows* $x - \left(x^{e}\right)^{d} = k \cdot p$ *and thus* $x - \left(x^{e}\right)^{d} \neq 0$ *which implies* $\gcd\left(x - (x^{e})^{d}, n\right) = p$ *and thus the modulus can be factored.*

6.3 FURTHER EXERCISES

1. Given the RSA encryption below with the corresponding modulus and exponent, find the encrypted message assuming that encryption was performed without padding.

n=
87166641318910733092980604362223878083629567867863418669374287834553659623916739172495744915952292070842977414645571321982290863656526
04590297378403184129

e=3

c=
13758655830109826186323085294233712718214385779809229271241303968779258635878271228868750245705568591220644581536 31

2. Given the RSA key-pair below find the factorization of the modulus.

n=

507631363489941354012053635005103431298761937877891150464742093854474651771103149011552842042731947927440738905825389749855711091316030280174187427760 8327,

e=3

d=

338420908993294236008035756670068954199174625251927433643161395902983101180725922665578612505088772792127471975198610416203780080764152234820737658337 9547

3. The following fact is considered an interesting property of the RSA, although we do not know the sum of the two factors of the modulus, i.e., $p+q$, we can compute the value of $x^{p+q} \bmod n$. Figure out how this is possible and compute this value for the numbers below.

n=

107006465856808858485205037352998524788658374387098151389928598832499554989162878572336274986066578667635927883395959219436274120529041619352017809284 78603,

x=

713376439045392389901366915686656831924389162580654342599523992216663699927738725319404850576734092459806416930413621058180990651121616876231863081831 1867

4. Factor the following integer, knowing that it is the power of a prime number. What is the expected number of steps to factor an integer of this form?

```
n=
14121216559045592723913725470284552915893297299545955512586695122770
93167352564280937489975075959990219486112359021551595669088036722 3
```

n=
1412121655904559272391372547028455291589329729954595551258669512277
0931673525642809374899750759599902194861123590215515956690880367223
6782701780153260648702410644513576680061002271472311778912389401527
8870040434452846004485093642675885009807658579541139272020261525991
6568029436599814044031229151775310358906532007112584154431330139440
8906580430629631327415853437044184526066718512464557009387552200433
0140817631416034869890537888261433693978718361566731421862575341925
9203124994887398592090289570466328291725708474859718918318673622960
749

5. To speed-up verification time for multiple RSA signatures, rather than verifying each signature independently, one can check the following equality: $\left(\prod_{i=1}^{k} s_i\right)^e = \prod_{i=1}^{k} h(m_i)$ (this is called batch verification). This method is fast as it requires a single modular exponentiation, in contrast to k exponentiations (and indeed modular exponentiation is the most expensive computational step in verifying signatures). However, there is a problem with this method: show that given multiple signatures $\{s_1, s_2, \dots, s_k\}$ corresponding to a set of messages $\{m_1, m, \dots, m_k\}$ one can produce a fake set of signatures that passes the batch verification test but no signature will hold for any of the messages in particular.

6. Prove the equivalence between the following computational problems: RSA-Key, computing Euler-Phi and Integer Factorization.

Note. Since there is no method in the *Java.BigInteger* class for computing integer square roots, you may recycle the naive code below.

```
//recursively searches for the sqr root of a in interval [left, right]
private static BigInteger NaiveSquareRootSearch(BigInteger a, BigInteger left,
BigInteger right)
   {
      // fix root as the arithmetic mean of left and right
      BigInteger root = left.add(right).shiftRight(1);
      // if the root is not between [root, root+1],
     //is not an integer and root is our best integer approximation
if(!((root.pow(2).compareTo(a)                    ==                    -
1)&&(root.add(BigInteger.ONE).pow(2).compareTo(a) == 1))){
         if (root.pow(2).compareTo(a) == -1) root = NaiveSquareRootSearch(a, root,
right);
         if (root.pow(2).compareTo(a) == 1) root = NaiveSquareRootSearch(a, left,
root);
      }
      return root;
   }

public static BigInteger SquareRoot(BigInteger a)
{
   return NaiveSquareRootSearch(a, BigInteger.ZERO, a);
}
```

Chapter 7. CRYPTOGRAPHY IN JAVA: SYMMETRIC AND ASYMMETRIC ENCRYPTIONS, PASSWORD BASED KEY-DERIVATIONS

The first subject of this section is understanding how encryption functions work in Java. Compared to .NET encryption is done a bit differently but it is not at all hard to do and more, there are external libraries, e.g., Bouncy Castle Crypto APIs, which have extensive support for many encryption functions that are not available in .NET. Moreover, the entire code is open source! For this reason you may prefer to work in Java, while for simplicity you may choose .NET.

Another subject that we reach in this section is how to generate keys. Password based key derivations and randomness are essential tools for generating cryptographic keys. The security of any cryptosystem ultimately depends on the randomness of the key, if the key is easy to guess, then the cryptosystem is trivially broken. Both these primitives are also available in the .NET framework, as well as the encryption primitives that we discussed in .NET have instances in Java.

7.1 SYMMETRIC AND ASYMMETRIC ENCRYPTION: AES, DES AND RSA

According to the Java SE (Standard Edition) documentation, see http://docs.oracle.com/javase/, the following algorithms are supported by the ***Cipher*** class:

- AES/CBC/NoPadding (128)
- AES/CBC/PKCS5Padding (128)
- AES/ECB/NoPadding (128)
- AES/ECB/PKCS5Padding (128)
- DES/CBC/NoPadding (56)
- DES/CBC/PKCS5Padding (56)
- DES/ECB/NoPadding (56)
- DES/ECB/PKCS5Padding (56)
- DESede/CBC/NoPadding (168)
- DESede/CBC/PKCS5Padding (168)
- DESede/ECB/NoPadding (168)
- DESede/ECB/PKCS5Padding (168)

- RSA/ECB/PKCS1Padding (1024, 2048)
- RSA/ECB/OAEPWithSHA-1AndMGF1Padding (1024, 2048)
- RSA/ECB/OAEPWithSHA-256AndMGF1Padding (1024, 2048)

Key sizes are available in parentheses, the mode of operation and padding are also specified. This is not much more support when compared to the .NET framework, we have the same DES, 3DES, AES and RSA suite. Fortunately, there are also many public extensions of Java so there are many others cryptographic functions, modes of operations or paddings supported by external implementations. One of the leading packages is the **Bouncy Castle Crypto package**, see https://bouncycastle.org/specifications.html , which has extensive support for many other constructions, e.g., IDEA, Serpent, RC4, in its ***Cipher*** class. There is clearly much more support for cryptography in Java than .NET, but it is also true that for regular applications it is unlikely that you will need more than the standard AES, 3DES and RSA.

To perform encryption and decryptions you first need to add some imports required for the objects that you are going to use as well as for the exceptions that are going to be thrown. These are summarized in Table 1. Besides the ***Cipher*** class from which all cryptosystems derive, you need to handle keys with ***Key, KeyPair*** and ***KeyPairGenerator*** classes then to randomly generate them with the ***SecureRandom*** class, please refer to the online documentation for more information.

```
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;
import java.security.SecureRandom;
import javax.crypto.Cipher;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.ShortBufferException;
```

```
import javax.crypto.spec.SecretKeySpec;
import javax.rmi.CORBA.Util;
```

Table 1. Some imports required in Java for the encryption

To perform encryption the paradigm is a bit different to that of .NET but not necessarily harder while the result is the same. To initialize a cipher object with a particular encryption scheme you will simply call a new instance with:

```
Cipher myAES = Cipher.getInstance("AES/ECB/NoPadding");
```

Then you will have to initialize this to work either in encryption or decryption mode as follows:

```
myAES.init(Cipher.ENCRYPT_MODE, myKey);
myAES.init(Cipher.DECRYPT_MODE, myKey);
```

Encryption and decryption work by simply updating the input with the plaintext (or ciphertext in case of decryption) and then calling the ***doFinal*** method for the remaining blocks (if any). The ***doFinal*** method returns the number of bytes stored in the buffer and the same is done by the ***update*** method. This is all summarized in Table 2.

```
byte[] keyBytes = new byte[16];
// declare secure PRNG
SecureRandom myPRNG = new SecureRandom();
// seed the key
myPRNG.nextBytes(keyBytes);
// build the key from random key bytes
SecretKeySpec myKey = new SecretKeySpec(keyBytes, "AES");
```

```
// instantiate AES object for ECB with no padding

Cipher myAES = Cipher.getInstance("AES/ECB/NoPadding");

// initialize AES objecy to encrypt mode

myAES.init(Cipher.ENCRYPT_MODE, myKey);

// initialize plaintext

byte[] plaintext = new byte[16];

//initialize ciphertext

byte[] ciphertext = new byte[16];

// update cipher with the plaintext

int cLength = myAES.update(plaintext, 0, plaintext.length, ciphertext,
      0);

// process remaining blocks of plaintext

cLength += myAES.doFinal(ciphertext, cLength);

// print plaintext and ciphertext

System.out.println("plaintext:                   "                  +
    javax.xml.bind.DatatypeConverter.printHexBinary(plaintext));

System.out.println("ciphertext:                  "                  +
    javax.xml.bind.DatatypeConverter.printHexBinary(ciphertext));

// initialize AES for decryption

myAES.init(Cipher.DECRYPT_MODE, myKey);

// initialize a new array of bytes to place the decryption

byte[] dec_plaintext = new byte[16];

cLength = myAES.update(ciphertext, 0, ciphertext.length, dec_plaintext,
    0);

// process remaining blocks of ciphertext

cLength += myAES.doFinal(dec_plaintext, cLength);

// print the new plaintext (hopefully identical to the initial one)

System.out.println("decrypted:                   "                  +
    javax.xml.bind.DatatypeConverter.printHexBinary(dec_plaintext));
```

Table 2. Example of AES encryption and decryption in Java

For asymmetric encryption or decryption the procedure is similar. The only distinction is that you now have to deal with two keys: a public and private one. For this reason you now have a ***KeyPair*** object to store the pair of keys, you ***Key*** objects to store individually the private or public key (when needed for wither decryption or encryption) and of course the ***KeyPairGenerator*** object to generate these keys. Table 3 summarizes the source code for the procedures.

```
// get a Cipher instance for RSA with PKCS1 padding
Cipher myRSA = Cipher.getInstance("RSA/ECB/PKCS1Padding");
// get an instance for the Key Generator
KeyPairGenerator myRSAKeyGen = KeyPairGenerator.getInstance("RSA");
// generate an 1024 bit key
myRSAKeyGen.initialize(1024, myPRNG);
KeyPair myRSAKeyPair= myRSAKeyGen.generateKeyPair();
// store the public and private key individually
Key pbKey = myRSAKeyPair.getPublic();
Key pvKey = myRSAKeyPair.getPrivate();
// init cipher for encryption
myRSA.init(Cipher.ENCRYPT_MODE, pbKey, myPRNG);
// encrypt, as expected we encrypt a symmetric key with RSA rather than
    a file or some longer stream which should be encrypted with AES
ciphertext = myRSA.doFinal(keyBytes);
// init cipher for decryption
myRSA.init(Cipher.DECRYPT_MODE, pvKey);
// decrypt
plaintext = myRSA.doFinal(ciphertext);
System.out.println("plaintext:                    "                +
    javax.xml.bind.DatatypeConverter.printHexBinary(plaintext));
System.out.println("ciphertext:                   "                +
    javax.xml.bind.DatatypeConverter.printHexBinary(ciphertext));
System.out.println("keybytes:                     "                +
    javax.xml.bind.DatatypeConverter.printHexBinary(keyBytes));
```

Table 3. Example of RSA encryption and decryption in Java

7.2 GENERATING KEYS: PASSWORD BASED KEY DERIVATION

Humans are not at all efficient in remembering, e.g., storing in mind, cryptographic random keys. But it happens that humans are better in remembering passwords or even longer sentences if these have some sense, i.e., passphrases. The issue with these is that they are generally incompatible with the format required for a cryptographic key. For example, an AES key has exactly 128 bit, but imagine that password can have 18 characters which require 144 bits for storing. The 18 character passwords have little chances in having 128 bits of entropy, i.e., being more secure than 128 bits picked at random, but is clearly easier to remember. If we truncate the 18 character password to 16 characters it will fit the 128 bits of the ley but it is a pity to lose the additional bits of entropy. Password based key derivation (PBKD) is here to help.

The main idea behind PBKD is to use a hash function in order to get an output of fixed size. But besides this hash function there are two more ingredients which you already met in the section dedicated to the UNIX password authentication system:

i) Salts, which and are used to prevent off-line guessing attacks,
ii) Iterations, which are used to make testing for each password more intensive and to hinder the adversary.

Both the salt and the iterations value are public and they not need to be kept secret.

The example provided in Table 1 shows how to generate a 128 bit key for AES by using a fixed password, a randomly generated salt and a fixed number of iterations. The number of iteration makes the key harder to crack by an adversary. The point is that a user will generate this key rarely, e.g., for each login, so waiting 10.000 iterations can take milliseconds which go unnoticeable. For an adversary however, the same procedure must be repeated for each password it tries, thus hindering the exhaustive search.

```
char[] password = "short_password".toCharArray();
byte[] salt = new byte[16];
int iteration_count = 10000;
```

```
int key_size = 128;
// set salt values to random
myPRNG.nextBytes(salt);


// initialize key factory for HMAC-SHA1 derivation
SecretKeyFactory                       keyFactory                      =
      SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
// set key specification
PBEKeySpec  pbekSpec  =  new  PBEKeySpec(password,  salt,  iteration_count,
      key_size);
// generate the key
SecretKey        myAESPBKey        =        new        SecretKeySpec(
      keyFactory.generateSecret(pbekSpec).getEncoded(), "AES");
// print the key
System.out.println("AES             key:             "             +
      javax.xml.bind.DatatypeConverter.printHexBinary(myAESPBKey.getEnc
      oded()));
```

Table 3. Example of password based key derivation for 128 bit AES with HMAC-SHA1 from password, random salt and 10000 iterations

7.3 EXERCISES

1. Write a program that performs encryption in CBC mode then in OFB and CFB by using a key that is generated from a user's password. Please remember to correctly set the IVs.

2. Write a program that derives keys from passwords and displays the computational time required for generating the password and the computational time required by an adversary to break the password by considering l iterations for password generation and passwords of k bit entropy.

FURTHER REFERENCES

You may find it useful to consult the following references for cryptography in .NET and Java:

[1] David Hook, ***Beginning Cryptography with Java***, Wiley Publishing, ISBN-13: 978-0764596339, ISBN-10: 0764596330, 2005

[2] Peter Thorsteinson, G. Gnana Arun Ganesh, ***.NET Security and Cryptography***, Prentice Hall, ISBN-13: 007-6092021490, ISBN-10: 013100851X, 2003.

[3] Jason R. Weiss, ***Java Cryptography Extensions: Practical Guide for Programmers***, Morgan Kaufmann, ISBN-13: 978-0127427515, ISBN-10: 0127427511, 2004.

[4] *****, Microsoft Developer Network (MSDN), https://msdn.microsoft.com

[5] *****, ORACLE Java Documentation , http://docs.oracle.com/javase/8/