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*.

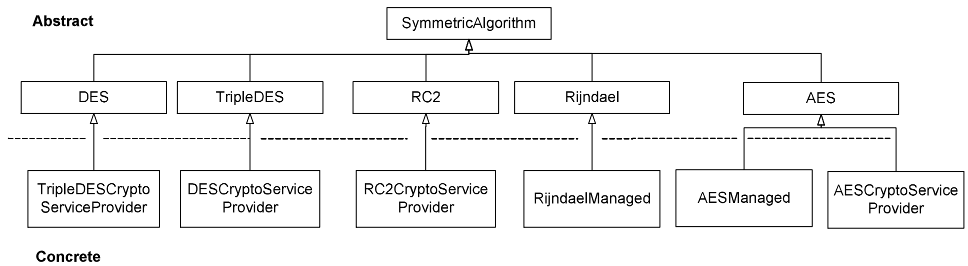


Figure 1. Symmetric encryption algorithms in .NET

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)

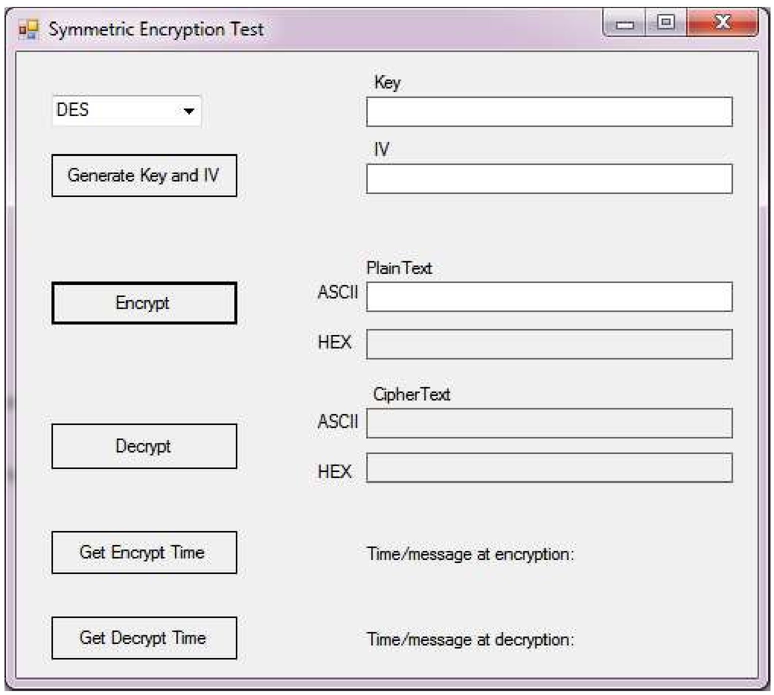
```
// Check arguments.
if (plainText == null || plainText.Length <= 0)
                throw new ArgumentNullException("plainText");
if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
if (IV == null || IV.Length <= 0)
```

```
                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;
        }
    }
```