Practical experience with the .NET cryptographic API

Cédric Boon, Pieter Philippaerts, Frank Piessens

DistriNet Research Group, Department of Computer Science
Katholieke Universiteit Leuven, Celestijnlaan 200A, B-3001 Leuven, Belgium

Abstract

When a vulnerability is discovered in a cryptographic algorithm, or in a specific implementation of that algorithm, it is important that software using that algorithm or implementation is upgraded quickly. Hence, modern cryptographic libraries such as the Java Cryptographic Architecture and Extensions (JCA/JCE) and the .NET crypto libraries are designed to be extensible with new algorithms. In addition, they also support algorithm and implementation independent use. Software written against these libraries can be implemented such that switching to a new crypto algorithm or implementation requires very little effort.

This paper reports on our experiences with the implementation of a number of extensions to the .NET cryptographic framework. The extensions we consider are smart card based implementations of existing algorithms. We evaluate the extensibility of the libraries, and the support for implementation independence. We identify several problems with the libraries that have a negative impact on these properties, and we propose solutions.

The main conclusion of the paper is that extensibility and implementation independence can be substantially improved with only minor changes. These changes maintain backwards compatibility for client code.

Index Terms

cryptography, .NET Framework, extensibility, algorithm independence, implementation independence

I. Introduction

A. Motivation

When Microsoft released the first version of the .NET Framework in 2002, they included a newly designed cryptographic API, built from the ground up. Like with the rest of the .NET runtime, one of the main concerns during the design of the architecture was to develop an object oriented system that is future-proof and ensures extensibility by end users. This is reflected in the class structure by modularizing the different algorithms, and applying standard object oriented design principles.

An additional, crypto-related, requirement was to also ensure algorithmic independence. Cryptography is a fast moving science where implementations of algorithms are regularly broken [1], [3], [11], [12], or where even entire classes of cryptographic algorithms are defeated [10], [13], [14]. This has as a consequence that secure applications must be able to quickly change from one algorithm to another. Having a cryptographic architecture that has been designed to allow this can be beneficial.

Even though the design goes a long way towards achieving these goals, some optimizations can be made. This paper describes some of the shortcomings of the .NET cryptographic architecture and proposes minimal changes to solve these problems. As the .NET Framework is used by numerous applications, a top priority will be to maintain backwards compatibility. Throughout the paper, the example of a smart card library will be used to expose the various problems with the API. It also provides a compelling case study, as smart card usage is increasing worldwide and will eventually be the cornerstone of many secure systems.

B. Organization of this paper

In the remaining part of this section, a short introduction to the .NET cryptographic framework is given. Readers who are already familiar with the cryptographic structure of .NET can skip this and start with section II. Part I-C of this section introduces the basics of the .NET crypto framework, whereas part I-D describes the particularities of extending the architecture with new Cryptographic Service Providers (CSP) for smart cards.

Sections II through V detail the experiences and problems with extending the framework. Each section explores specific classes of algorithms in the API. A section is composed of a sketch of what the current implementation looks like, a description of the problems that were encountered, and the proposed solution to counter these issues.

Finally, section VI summarizes the observations.

C. Introduction to the .NET cryptographic API

Due to the volatile nature of the cryptographic domain, the architecture of a cryptographic API must be resilient to change. Algorithms, and implementations of algorithms, come and go on a regular basis. Additionally, because it's important to be able to quickly change an application to use a different algorithm, the API must also support some easy means of substituting old cryptographic service providers with new ones. This leads to the observation that two architectural qualities are key for a cryptographic framework:

a) Extensibility: New algorithms are regularly introduced to counter the loss of broken algorithms, so the framework must have hooks to extend the base system to support other algorithms. Moreover, one algorithm can have multiple implementations. When new optimizations are found, for example, older implementation may be replaced with new, faster, implementations.

b) Algorithmic independence: One certainty in cryptography is that algorithms get broken. When this happens, applications depending on this algorithm must be able to quickly switch to another, more secure algorithm. Hence, some kind of independence of the algorithm must be achieved.

The .NET cryptographic API uses an inheritance-based model to achieve these goals. The architecture consists of three layers:

- **The engine classes:** Engine classes represent a group of algorithms, like symmetric algorithms (the abstract *SymmetricAlgorithm* class), asymmetric algorithms (the abstract *AsymmetricAlgorithm* class), or hash algorithms (the abstract *HashAlgorithm* class). These abstract classes feature a *Create* method method that returns a default implementation of the default algorithm for the concerning engine class.
- **The algorithm classes:** Algorithm classes represent a specific algorithm. They are abstract and derive from the abstract engine classes. The *DES* algorithm class, which inherits from the *SymmetricAlgorithm* engine class, and the *SHA1* algorithm class, which inherits from the *HashAlgorithm* engine class, are examples of such abstract algorithm classes. As with the engine classes, algorithm classes define a *Create* method that returns a default implementation of the specific algorithm.
- **The implementation classes:** Implementation classes contain the actual implementation of the algorithms. These classes are concrete, and derive from the abstract algorithm classes. The .NET framework ships with a number of concrete implementations, such as the *DESCryptoServiceProvider* class as an implementation of the *DES* algorithm, the *SHA1CryptoServiceProvider* class as an implementation of the *SHA1* algorithm, ...

Figure 1 shows the hierarchy of the symmetric and asymmetric algorithms. The names of the classes in the bottom of the hierarchy, the implementation classes, always end with 'Managed' or 'CryptoServiceProvider. This difference in naming indicates how they are implemented. 'Managed' classes are written completely in .NET managed code. Hence, the full cryptographic algorithm was reimplemented when the .NET cryptographic framework was built. Their 'CryptoServiceProvider' counterparts on the other hand are wrappers around existing code. They call the *native Windows CryptoAPI* (or short: *CryptoAPI*). This is the cryptographic API that's been present in Windows long before .NET existed. It is typically used by so called native (often C/C++) applications.

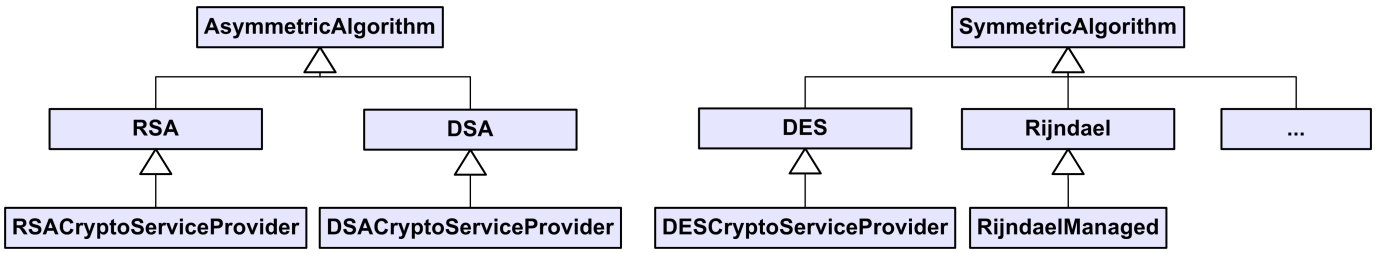

Figure 1. The structure of the .NET cryptographic API for asymmetric algorithms and hash algorithms.

Developers are encouraged to work with the engine classes. The *Create* method of these classes will return a default implementation of the default algorithm for the engine class. The .NET Framework provides the ability to modify the default algorithms and implementations by adjusting settings in the cryptographic configuration file. If an application uses only the abstract engine classes, changing the cryptographic algorithms used by that application can be achieved by modifying the cryptographic configuration file, without changing the implementation of the application or having to recompile the application.

The cryptographic configuration is used to resolve specific algorithm implementations from algorithm names. This makes it possible for system administrators to specify the default algorithms and implementations on a particular machine. Listing 1 illustrates how the cryptographic configuration can be modified to replace the default implementation of the *SHA1* algorithm by a custom implementation *MySHA1HashClass*.

```
<configuration>
 <!-- Other configuration settings. -->
 <mscorlib>
  <cryptographySettings>
   <cryptoNameMapping>
    <cryptoClasses>
     <cryptoClass MySHA1Hash="MySHA1HashClass,
      MyAssembly
      Culture='en',
      PublicKeyToken=a5d015c7d5a0b012,
      Version=1.0.0.0"/>
    </cryptoClasses>
    <nameEntry name="SHA1"
     class="MySHA1Hash"/>
    <nameEntry name="System.Security.
     Cryptography.SHA1"
     class="MySHA1Hash"/>
    <nameEntry name="System.Security.
     Cryptography.HashAlgorithm"
     class="MySHA1Hash"/>
   </cryptoNameMapping>
  </cryptographySettings>
 </mscorlib>
</configuration>
```

Listing 1. Redefinition of the default implementation for SHA1.

D. Custom cryptographic service providers for smart cards

Smart cards have been around for a while, mainly used in the banking and telecommunication sector. Only recently, however, the interest in smart cards has increased tremendously, with applications for cell phones [7], e-government [4], authentication [6], ... Smart cards offer a substantial security improvement over traditional magnetic stripe cards, because they contain a small processor that can execute cryptographic operations on data that is contained within the card [5]. This means that secret data, such as the private part of an asymmetric key or a secret symmetric key, doesn't have to leave the card to compute the output of some cryptographic algorithm.

Version 3.5 of the .NET Framework does not provide direct support to perform cryptographic operations on smart cards. While it is possible to initialize an *RSACryptoServiceProvider* or a *DSACryptoServiceProvider* class with a *CspParameters* instance that specifies a smart card cryptographic service provider from the native Windows CryptoAPI (see listing 2), this requires a custom smart card CSP to be installed on the target machine. Furthermore, the *DSACryptoServiceProvider*, the *RSACryptoServiceProvider* and the *RNGCryptoServiceProvider* are the only CSPs that accept such a *CspParameters* instance. Using smart cards to perform other operations, such as symmetric cryptography or hashing, requires a lot of implementation work and a thorough understanding of the smart card communication protocol.

```
CspParameters csp = new CspParameters(1,
   "Schlumberger Cryptographic Service
    Provider");
csp.Flags =
   CspProviderFlags.UseDefaultKeyContainer;
RSACryptoServiceProvider rsa =
   new RSACryptoServiceProvider(csp);
```

Listing 2. Initialization of the RSACryptoServiceProvider class with CspParameters to indicate that a smart card CryptoAPI CSP should be used.

To ease the development of .NET applications that rely on smart cards for cryptographic operations, we developed a number of custom CSPs based on the *ISO 7816-4* [9] and *ISO 7816-8* [8] standards. Care has been taken to fit the smart card CSPs into the current .NET cryptographic framework, to simplify the end-user's experience. The problems with the .NET crypto framework are mainly based on our experience with the integration of these smart card CSPs. Note however that the problems described in this paper are not specific to smart cards, but are indications of imperfections with respect to the cryptographic model's extensibility.

II. Asymmetric Algorithms

A. Asymmetric Algorithms in the .NET Framework

The engine class for asymmetric algorithms in the .NET Framework is the abstract *AsymmetricAlgorithm* class. All asymmetric algorithm classes must inherit from this engine class. Version 3.5 of the .NET Framework ships with cryptographic service providers for RSA, DSA, and the elliptic curve variants of Diffie-Hellman and DSA.

The engine class defines members for the creation of (default or named) asymmetric algorithm classes, for the reconstruction of an *AsymmetricAlgorithm* object from an XML string, and for the serialization of an *AsymmetricAlgorithm* object to an XML string.

On the second level of the hierarchy, classes inherit from the engine class and represent all implementations of a particular algorithm. The abstract *RSA* class, for example, inherits from the engine class for asymmetric algorithms, and represents an algorithm class for all implementations of the RSA algorithm. The *RSA* class defines the additional abstract members *EncryptValue* and *DecryptValue*. When overridden in a derived class, these methods perform *raw* RSA encryption and decryption and return the result. In this context, *raw* means that the input data should be encrypted or decrypted without performing any padding.

Padding is an essential step when using some cryptographic algorithms. Inappropriate padding can introduce potential vulnerabilities and substantially reduce the effort needed to perform attacks on cryptographic algorithms. Such attacks have been demonstrated in [2] for *PKCS#1 1.5*[1] padding.

The .NET Framework provides an elegant solution to cope with different padding schemes for RSA and DSA by means of formatters. A formatter accepts an *AsymmetricAlgorithm* as a parameter and takes care of padding operations before passing on the modified data for encryption or signing. Specific formatters accept only specific children of *AsymmetricAlgorithm*. This architecture allows for different padding schemes to be used with one *AsymmetricAlgorithm*, and provides an extensible base for future padding schemes. Version 3.5 of the .NET Framework defines the following abstract base formatters:

- **AsymmetricKeyExchangeFormatter:** Derived classes perform the padding operations needed for asymmetric encryption.
- **AsymmetricKeyExchangeDeformatter:** Derived classes unpad the data from an asymmetric decryption, and verify the correctness of the padding.
- **AsymmetricSignatureFormatter:** Derived classes perform the padding operations needed for the creation of a digital signature.
- **AsymmetricSignatureDeformatter:** Derived classes perform the padding operations needed for the verification of a digital signature.

[1]PKCS#1 1.5: Public-Key Cryptography Standards #1 version 1.5

Concrete implementations for *PKCS#1 1.5* and *OAEP*[2] padding for RSA operations, as well as a formatter and deformatter for DSA operations are available in all versions of the .NET Framework.

As previously mentioned, the potential extension possibilities are one of the big advantages of the use of formatters. When weaknesses in current padding schemes are found, new formatters (like e.g. RSA *PSS*[3] signature formatters) could be defined. Furthermore, upgrading an existing application with a new padding scheme is as simple as changing one line of code.

B. RSA

1) Identified problems: Version 3.5 of the .NET Framework relies on the *CryptoAPI* to implement the RSA cryptographic service provider. It does not ship with an RSAManaged class. The *CryptoAPI*, however, does not expose *raw* RSA operations. An invocation of the API has to specify the padding scheme to be used. To fit the .NET RSA CSP in the model of formatters, the formatters handle instances of the *RSACryptoServiceProvider* class differently than other subclasses of the *RSA* class.

If the asymmetric algorithm passed to an RSA formatter *is not* an instance of the *RSACryptoServiceProvider* class, padding is performed by the formatter, and the *EncryptValue*/*DecryptValue* methods are invoked. The *EncryptValue*/*DecryptValue* methods always perform raw RSA operations.

The *RSACryptoServiceProvider* does not implement the *EncryptValue*/*DecryptValue* methods. Instead, additional methods are defined that delegate the operations to the *CryptoAPI*. If the asymmetric algorithm passed to an RSA formatter *is* an instance of the *RSACryptoServiceProvider* class, no padding is performed by the formatter. Instead, one of the additional methods of the *RSACryptoServiceProvider* is invoked, and the result is returned as received by the *CryptoAPI*.

While the model retains the elegance of the API, version 3.5 of the .NET Framework incorrectly assumes that the *RSACryptoServiceProvider* class is the only subclass of *RSA* that does not support raw RSA. By deriving from the abstract *RSA* class, additional implementations of RSA can be provided by third parties. As mentioned in section I, an RSA CSP for smart cards has been used for the evaluation of the cryptographic framework. Performing RSA operations on a smart card has the advantage that private keys never have to leave the smart card, offering enhanced protection for the storage of this private data. The smart card RSA CSP inherits from the abstract *RSA* class, and can be passed to the RSA formatters to perform the required padding operations. However, the smart card standards[4] allow manufacturers to choose whether padding operations are performed on-card or off-card. The smart card CSP supports both options, through analogy with the *RSACryptoServiceProvider* class. Unfortunately, when a smart card RSA CSP is passed on to an RSA formatter, there is no way to specify whether the RSA CSP does or does not support raw RSA. Only the *RSACryptoServiceProvider* class can bypass the formatter's padding operations. This introduces problems when using formatters with smart cards that only support on-card padding.

One could argue that this inconsistency could be solved in the smart card CSP itself. The *Encrypt* method could remove any padding added by a formatter, while the *Decrypt* method could add the padding removed by the smart card. However, this design would introduce additional problems. Firstly, padding information would be introduced in the smart card RSA CSP class, which undermines the use of formatters. Secondly, the *Decrypt* method should perform padding on the result before returning it to a deformatter. But as no information on the deformatter type is available in the smart card CSP itself, the correct padding scheme cannot be deduced.

Previous paragraphs clearly emphasize the need for formatters to support both *raw* and *padded* RSA operations on more than just the *RSACryptoServiceProvider* class. The following section describes the modifications required to achieve this support, while retaining backwards compatibility.

2) Suggested solution: The suggested solution consists of a *pull up* of several methods from the *RSACryptoServiceProvider* class to the *RSA* class. The methods of the *RSACryptoServiceProvider* that are required by the formatters for *padded* operations are:

- public byte[] Encrypt(byte[] rgb, bool fOAEP)
- public byte[] Decrypt(byte[] rgb, bool fOAEP)
- public byte[] SignHash(byte[] rgbHash, string str)

[2]OAEP: Optimal Asymmetric Encryption Padding

[3]PSS: Probabilistic Signature Scheme

[4]Smart card standards: ISO 7816

- public bool VerifyHash(byte[] rgbHash, string str, byte[] rgbSignature)

To support the decision process of the formatters, an additional method should be inserted in the *RSA* class:

- public virtual bool SupportsRaw()

In order to preserve compatibility with existing applications, and to promote extensibility of the proposed solution, a number of changes to the pulled up methods are required:

- The pulled up methods should be marked *virtual*, to allow for dynamic binding.
- An implementation for these methods should be provided, to preserve compatibility with existing subclasses that do not implement these methods. A suggested implementation is throwing a *NotImplementedException*. This is analog to how the current .NET framework solves similar issues.
- The boolean parameter in the *Encrypt/Decrypt* methods should be changed to a more general parameter, like a *String*. This allows extensions with padding types other than *PKCS#1 1.5* and *OAEP*. An Object Identifier (Oid) could also be used, if Oids for padding types are registered.

The final changes to the *RSA* class are shown in listing 3.

```
public virtual byte[] Encrypt(
    byte[] rgb,
    string padding)
{
    throw new NotImplementedException();
}
public virtual byte[] Decrypt(
    byte[] rgb,
    string padding)
{
    throw new NotImplementedException();
}
public virtual byte[] SignHash(
    byte[] rgbHash,
    string str,
    string padding)
{
    throw new NotImplementedException();
}
public virtual bool VerifyHash(
    byte[] rgbHash,
    string str, byte[] rgbSignature,
    string padding)
{
    throw new NotImplementedException();
}
public virtual bool SupportsRaw()
{
    return true;
}
```

Listing 3. Methods that should be added to the *RSA* class.

The *RSACryptoServiceProvider* class should implement the modified *Encrypt/Decrypt* methods by delegating the calls to the original methods, supplied with the appropriate boolean parameter. The implementation of the formatters should be modified to benefit from the new structure, as shown for the *RSAPKCS1KeyExchangeFormatter* in listing 4.

An interesting question is whether the methods should be pulled up to the *RSA* class, or to the *AsymmetricAlgorithm* class. The *DSACryptoServiceProvider* class defines the same *SignHash* and *VerifyHash* methods as the *RSACryptoServiceProvider* class. Furthermore, the *CreateSignature* and *VerifySignature* methods that are used by the DSA formatters simply delegate the calls to the *SignHash* and *VerifyHash* methods. These facts are in favor of pulling the methods up to the level of the *AsymmetricAlgorithm* class.

```
if(_rsaKey.SupportsRaw())
{
    // Perform Padding
    data = PerformPadding(data);
    //Encrypt the padded data
    return _rsaKey.EncryptValue(data)
} else
{
    // Encryption and padding the unpadded
    // data
    return _rsaKey.Encrypt(data,"PKCS1")
}
```

Listing 4. Modification of the *RSAPKCS1KeyExchangeFormatter*.

The two elliptic curve asymmetric algorithms (*ECDiffieHellman* and *ECDSA*), however, do not share the same structure. Furthermore, the *DSA* and *DSACryptoServiceProvider* classes do not define *Encrypt* and *Decrypt* methods, and the DSA formatters do not experience the same problem the RSA formatters do, as there is no need for multiple padding schemes with DSA.

While pulling up the methods to the level of the abstract *AsymmetricAlgorithm* class could lead to a somewhat higher consistency in the class hierarchy, it will also significantly increase the complexity of the modifications and introduce redundant methods, without providing additional benefits for the RSA formatters. As such, it is not our recommendation to do this.

Under the assumption that the methods are pulled up to the level of the *RSA* class, it is important to show that the proposed modifications do not break the compatibility with existing applications:

- For users of the *RSACryptoServiceProvider* class, the only change is the addition of three methods, and the addition of an *override* keyword to two existing methods.
- For users of the abstract *RSA* class, the only change is an addition of five methods.
- For classes derived from the abstract *RSA* class, five more methods are available to override. As the *RSA* class provides concrete implementations for these methods, no additional implementation is required at the level of the derived class, which preserves compatibility with existing third party RSA subclasses.
- For third party classes deriving from the abstract *RSA* class and implementing methods with the same signature as the added ones, the existing methods will hide the added methods in the *RSA* base class, leaving the functionality unchanged.
- For users of the formatters, the new implementation would be semantically identical to the existing one for existing classes. As the *SupportsRaw* method would return a default value of *true* and would be overridden in the *RSACryptoServiceProvider* class to return *false*, the behavior of the formatters would be the same as provided by version 3.5 of the .NET Framework, for all existing RSA implementations.

C. Elliptic Curve Algorithms

The elliptic curve algorithm classes (*ECDiffieHellman* and *ECDsa*, as well as implementations of these algorithms (*ECDiffieHellmanCng* and *ECDsaCng*) were added in version 3.5 of the .NET Framework. Like the *RSACryptoServiceProvider*, the default implementation of *ECDiffieHellman* does not support *raw* operations. Whenever the Diffie-Hellman algorithm is used, some kind of *key derivation function* must be used to convert the raw output bits to a cryptographic key of the desired size. For the *ECDiffieHellmanCng* class, a *KeyDerivationFunction* property has to be set that specifies which derivation function should be used. This property is defined as an enum, for which the values *Hash*, *Hmac* and *Tls* are allowed. Depending on this property, the additional properties *HashAlgorithm* and/or *HmacKey* can be set. Listing 5 illustrates the use of the *ECDiffieHellmanCng* implementation class.

1) Identified problem: The elliptic curve Diffie-Hellman algorithm class, *ECDiffieHellman*, does not provide a method to set the different key derivation functions. While this enhances extensibility —new algorithms different from those defined in the enums of the default implementation can be used by overriding the algorithm class— it also reduces the implementation independency of the *ECDiffieHellman* class. An application that uses the

```
ECDiffieHellmanCng alice
  = new ECDiffieHellmanCng();
alice.KeyDerivationFunction
  = ECDiffieHellmanKeyDerivationFunction.Hash;
alice.HashAlgorithm = CngAlgorithm.Sha256;

ECDiffieHellmanCng bob
  = new ECDiffieHellmanCng();
bob.KeyDerivationFunction
  = ECDiffieHellmanKeyDerivationFunction.Hash;
bob.HashAlgorithm = CngAlgorithm.Sha256;

byte[] bobKey
  = bob.DeriveKeyMaterial(alice.PublicKey);
byte[] aliceKey
  = alice.DeriveKeyMaterial(bob.PublicKey);

AesCryptoServiceProvider aes
  = new AesCryptoServiceProvider();
aes.Key = aliceKey;
```

Listing 5. Use of the *ECDiffieHellmanCng* class.

ECDiffieHellmanCng class might not be able to easily switch to a different implementation, due to the lack of a standardized interface.

2) Suggested solution: The suggested solution consists of two steps. In the first step, the Elliptic Curve algorithms are extended with *formatters*, through analogy with the formatters for the *RSA* and *DSA* asymmetric algorithms. In a second step, the same changes to the formatters for *ECDiffieHellman* are applied as the changes mentioned in section II-B for the RSA formatters. This allows for the use of *formatters* with third party cryptographic service providers that, like the default implementation of the *ECDiffieHellman* algorithm, do not support *raw* operations.

The implementation of the *ECDsa* formatter and deformatter would be fairly straightforward. As the *ECDsa* algorithm class is very similar to the *DSA* algorithm class, *ECDsaSignatureFormatter* and *ECDsaSignatureDeformatter* classes can be implemented through analogy with the *DSASignatureFormatter* and *DSASignatureDeformatter* classes.

The implementation of *formatters* for the *ECDiffieHellman* algorithm would require one formatter/deformatter pair for each possible key derivation function. In the following paragraphs, we will elaborate on a formatter and deformatter for the *SHA1* derivation function.

An *ECDiffieHellmanSHA1KeyExchangeFormatter* would inherit from *AsymmetricKeyExchangeFormatter*. Through analogy with the *RSA formatter*, the *CreateKeyExchange* method could be defined to have an implementation as presented in listing 6.

The *ECDiffieHellmanSHA1KeyExchangeDeformatter* would inherit from the base class *AsymmetricKeyExchangeDeformatter*. Through analogy with the *RSA* deformatter, the *DecryptKeyExchange* method could be defined to have an implementation as presented in listing 7. Note that this would also require making the ECDiffieHellmanPublicKey class concrete instead of abstract. As the abstract ECDiffieHellmanPublicKey has an internal state that matches its purpose entirely, and the only abstract method is the *ToXmlString* method, this should not introduce any problems.

Formatters for other key derivation functions could be introduced in a similar way.

While introducing formatters for the Elliptic Curve algorithms would already be a step in the right direction, the implementation of the *ECDiffieHellman* formatters as presented in listings 6 and 7 introduces the same problem as identified in section II-B with the *RSA* formatters. More specifically, no third party CSPs that do not support *raw* operations would be usable with these formatters. The suggested solution is similar to the solution presented in section II-B2 for the *RSA* formatters:

```
if(this._key is ECDiffieHellmanCng){
 ((ECDiffieHellmanCng)_key).
  KeyDerivationFunction
  = ECDiffieHellmanKeyDerivationFunction.Hash;
 ((ECDiffieHellmanCng)_key).HashAlgorithm
  = CngAlgorithm.Sha1;
 return _key.PublicKey.ToByteArray();
}

byte[] rawData = _key.PublicKey.ToByteArray();
return SHA1.Create().ComputeHash(rawData);
```

Listing 6. A possible implementation of the *CreateKeyExchange* method for an *ECDiffieHellmanSHA1KeyExchangeFormatter*.

```
if(this._key is ECDiffieHellmanCng){
 ((ECDiffieHellmanCng)_key).
  KeyDerivationFunction
  = ECDiffieHellmanKeyDerivationFunction.Hash;
 ((ECDiffieHellmanCng)_key). HashAlgorithm
  = CngAlgorithm.Sha1;
 ECDiffieHellmanCngPublicKey publicKey
  = ECDiffieHellmanCngPublicKey.FromByteArray(
  rgb, CngKeyBlobFormat.EccPublicBlob);
 return _key. DeriveKeyMaterial(publicKey);
}

ECDiffieHellmanPublicKey publicKey
 = new ECDiffieHellmanPublicKey(rgb)
byte[] rawData
 = _key. DeriveKeyMaterial(publicKey);
return SHA1.Create().ComputeHash(rawData);
```

Listing 7. A possible implementation of the *DecryptKeyExchange* method for an *ECDiffieHellmanSHA1KeyExchangeDeformatter*.

- The introduction of a *"public virtual bool SupportsRaw()"* method at the level of the *ECDiffieHellman* algorithm class.
- The introduction of a *"public virtual bool SetKeyDerivationFunction(Oid algorithm, object parameters)"* at the level of the *ECDiffieHellman* algorithm class

Based on the *SupportsRaw* method, either the *SetKeyDerivationFunction* would be called (if the *ECDiffieHellman* implementation does not support raw operations), or the derivation function would be applied in the formatter itself (if the *ECDiffieHellman* implementation does support raw operations). The *Parameters* property of the asymmetric key exchange formatters could be used to pass on the parameters required for the key derivation function, such as a key in the case of an *hmac* derivation function.

The introduction of formatters for the Elliptic Curve algorithms does not break the backwards compatibility:

- For users of the *ECDiffieHellmanCng* class, the only change is the addition of two methods.
- For users of the abstract *ECDiffieHellman* class, the only change is the addition of two methods.
- For classes derived from the abstract *ECDiffieHellman* class, two more methods are available to override. As the abstract *ECDiffieHellman* class provides concrete implementations for these methods, no additional implementation is required at the level of the derived class, which preserves compatibility with existing third party ECDiffieHellman implementations.
- For third party classes deriving from the abstract *ECDiffieHellman* class and implementing methods with the same signature as the added ones, the existing methods will hide the added methods in the *ECDiffieHellman* base class, leaving the functionality unchanged.

III. Symmetric Algorithms

A. Symmetric in the .NET Framework

The base class for symmetric algorithms in the .NET Framework is the abstract *SymmetricAlgorithm* class. All implementations of symmetric algorithms must inherit from this base class. Version 3.5 of the .NET Framework provides cryptographic service providers for the AES, (Triple)DES, RC2 and Rijndael algorithms.

In contrast to the asymmetric algorithms, the class hierarchy of symmetric algorithms is more symmetrical (see tables I and II), with most of the operational methods concentrated in the abstract *SymmetricAlgorithm* base class. The engine class defines a *Key* property, which gets or sets a byte array that represents the key for the symmetric algorithm. This byte array represents a transparent key, because the interpretation of this data is fixed and known. For the asymmetric algorithms, methods for passing *transparent* keys are only defined at the second level in the class hierarchy. The abstract *RSA* class for instance defines the methods *ImportParameters* and *ExportParameters* to import or export an *RSAParameters* structure, a transparent representation of the keys for RSA. The class *AsymmetricAlgorithm*, however, also defines the methods *FromXmlString* and *ToXmlString*, which can be used to get or set an XML representation of the concrete asymmetric algorithm. The only requirement of these methods is that the input and output has a valid XML syntax. These methods can thus be used to get or set *opaque* keys, represented by an XML string. Opaque keys do not have a predefined data structure.

Table I
THE NUMBER OF NEW PUBLIC AND PROTECTED, NON-STATIC METHOD AND PROPERTY DEFINITIONS BY HIERARCHICAL LEVEL FOR SYMMETRIC ALGORITHMS.

	Methods	Properties
SymmetricAlgorithm	9	9
Aes	0	0
DES	0	0
RC2	0	1
Rijndael	0	0
TripleDES	0	0
AesCryptoServiceProvider	0	0
AesManaged	0	0
DESCryptoServiceProvider	0	0
RC2CryptoServiceProvider	0	1
RijndaelManaged	0	0
TripleDESCryptoServiceProvider	0	0

Table II
THE NUMBER OF NEW PUBLIC AND PROTECTED, NON-STATIC METHOD AND PROPERTY DEFINITIONS BY HIERARCHICAL LEVEL FOR ASYMMETRIC ALGORITHMS.

	Methods	Properties
AsymmetricAlgorithm	4	4
DSA	4	0
ECDiffieHellman	1	1
ECDsa	2	0
RSA	4	0
DSACryptoServiceProvider	8	3
ECDiffieHellmanCng	5	9
ECDsaCng	9	2
RSACryptoServiceProvider	11	3

One of the objectives of the cryptographic framework is to be able to easily substitute an algorithm by another one in the case that the first one is broken. Consider the code in listing 8. By using the *Create* method of the abstract *SymmetricAlgorithm* class, a default implementation of a symmetric algorithm will be returned. Version 3.5 of the .NET Framework will return an instance of the *RijndaelManaged* class. However, the framework allows overriding the default cryptographic service providers by means of the cryptographic configuration.

```
SymmetricAlgorithm symmetricAlgorithm =
    SymmetricAlgorithm.Create();
symmetricAlgorithm.Key = myKey;
```

Listing 8. Creation of a symmetric algorithm.

The .NET cryptographic configuration is located in the *<cryptographySettings>* section of the *machine.config* file. This file can be found in the *"%runtime install path%\Config"* directory.

An application using symmetric algorithms via the *Create* method of the *SymmetricAlgorithm* class can switch between symmetric algorithms by changing the cryptographic configuration, without any modifications to the source code. Should the default algorithm be broken, no changes to the source code are required to upgrade these applications to use a new algorithm.

B. Identified Problems

One of the advantages of using smart cards for symmetric algorithms is that the key never has to leave the smart card. Consider a system where data has to be encrypted using a symmetric algorithm. A random key could be generated on a smart card, and used in the encryption and decryption process. The key on the smart card could be made accessible to the smart card only, upon entry of a PIN. Once the data is encrypted using the smart card, it will only be decryptable with the same smart card.

As mentioned in section I, a number of symmetric CSPs (DES, 3DES and Rijndael) for smart cards have been used for the evaluation of the cryptographic framework. No key property has to be given to those cryptographic service providers, as the key is already available on the smart card itself. However, these cryptographic service providers need a number of other parameters, like a PIN (with adequate formatting parameters), an identifier representing the keyslot to be used on the smart card, ...

While it is possible to introduce *ImportParameters* and *ExportParameters* methods that accept opaque key material at the level of the implementation of the CSP itself, this prevents users from exploiting the advantages of the .NET cryptographic architecture, as a cast to the concrete implementation class would always be necessary to set the parameters. Having a mechanism to import opaque key material for symmetric algorithms on a more abstract level would therefore be beneficial to the developers using the framework.

C. Suggested solution

Through analogy with the abstract base class for asymmetric algorithms, the following methods could be defined in the abstract *SymmetricAlgorithm* engine class:

- public virtual void FromXmlString(string xmlString);
- public virtual abstract string ToXmlString();

These methods import and export opaque key material as an XML string. A default implementation in the abstract *SymmetricAlgorithm* base class could be used to get or set the key, the initialization vector, the cipher mode and the padding mode. Consider a modified version of listing 8 presented in listing 9.

```
SymmetricAlgorithm symmetricAlgorithm =
    SymmetricAlgorithm.Create();
symmetricAlgorithm.FromXmlString(
    myOpaqueXmlParameters);
```

Listing 9. Creation and initialization of a symmetric algorithm using opaque parameters.

In version 3.5 of the .NET Framework, listing 8 would have to be modified with a cast to the smart card based CSP to upgrade the application with smart card symmetric cryptography. In contrast, listing 9 would allow to upgrade the application through the cryptographic configuration, without any modifications to the application itself. This is under the assumption that the XML parameters can be retrieved from a store external to the application (e.g. a protected file).

For the asymmetric algorithms, the parameters needed by each algorithm are fairly different, which is why the *FromXmlString* and *ToXmlString* methods are defined as abstract in the *AsymmetricAlgorithm* base class, and only implemented at the level of the algorithm classes. As the class hierarchy of the symmetric algorithms concentrates a lot of the required parameters in the *SymmetricAlgorithm* class, however, the proposed *FromXmlString* and *ToXmlString* methods for symmetric algorithms could be marked *virtual*, and provided with a default implementation in the *SymmetricAlgorithm* engine class.

The modifications to the symmetric algorithms introduced in the previous paragraphs do not break compatibility with existing applications:

- For users of the *SymmetricAlgorithm* class, the only change is an addition of two methods.
- For classes derived from the *SymmetricAlgorithm* class, 2 more methods are available to override. As the *SymmetricAlgorithm* class provides concrete implementations for these methods, no additional implementation is required at the level of the derived class, which preserves compatibility with existing third party symmetric algorithm implementations.
- For third party classes deriving from the *SymmetricAlgorithm* class and implementing methods with the same signature as the added ones, the existing methods will hide the added methods in the *SymmetricAlgorithm* engine class, leaving the functionality unchanged.

IV. Keyed Hash Message Authentication Codes (HMAC)

A. HMACs in the .NET Framework

Figure 2 shows the class hierarchy for hash algorithms in the .NET cryptographic framework. The abstract *HMAC* class is the algorithm class from which all Hash-based Message Authentication Codes must derive. Version 3.5 provides cryptographic service providers for HMACs based on the MD5, SHA1, SHA2 and RIPEMD-160 hashes.

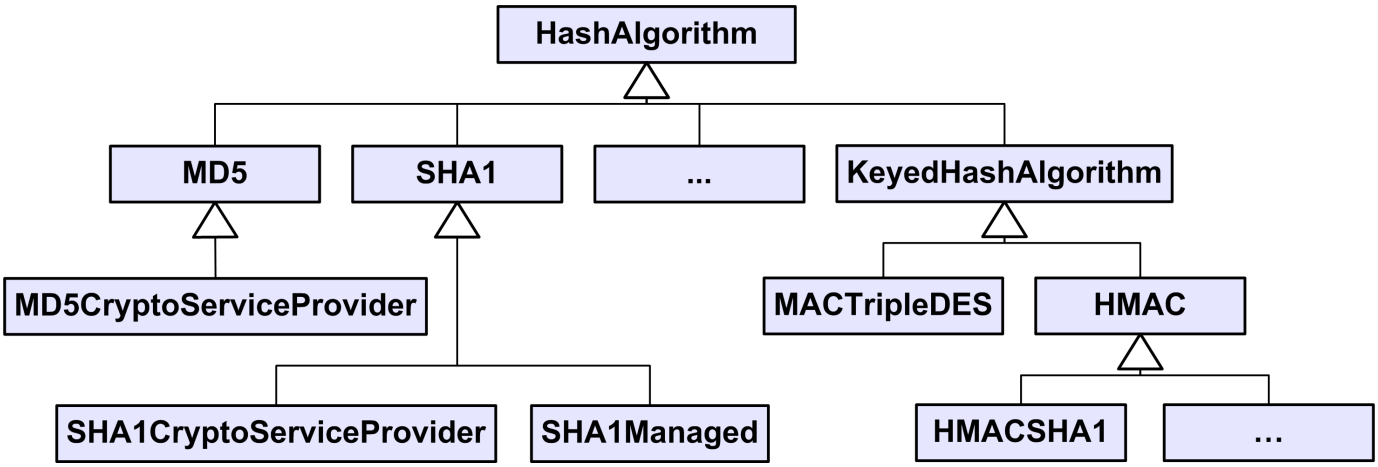

Figure 2. Class hierarchy for hash algorithms.

The HMAC algorithm is the same for all implementations of the abstract *HMAC* class. Only the hash function differs from implementation to implementation. For this reason, the abstract *HMAC* class contains most of the functionality, while classes derived from this base class mainly initialize the correct hash algorithm in this base class.

B. Identified Problems

As mentioned in section I, a SHA1 CSP for smart cards has been used for the evaluation of the cryptographic framework. In the context of HMACs, the CSP has been used to create a smart card HMAC-SHA1 CSP, by deriving from the *HMAC* class. SHA1 hashing operations are delegated to the smartcard, whereas the other operations required to calculate an HMAC are performed in software.

The problem with deriving from the *HMAC* class is that the methods used to implement the HMAC algorithm in this base class are marked as *internal*. Thus, deriving from the *HMAC* class does not support reusing the existing implementation in this class.

A number of ways exist to create a CSP that derives from the abstract *HMAC* base class. It is possible to re-implement the functionality of the base class in the derived class, by overriding the *HashCore* and *HashFinal* methods. However, this would violate the principles of object oriented design that try to promote the reuse of code. This is also why the additional abstraction of the *HMAC* class was introduced between the *KeyedHashAlgorithm* and the concrete implementations in the first place.

Another way to create a CSP deriving from the abstract *HMAC* base class would be to use the *HashName* property to set the required hash algorithms in the constructor of the deriving class. However, this requires that the assembly of the smart card based hash implementation is added in the global assembly cache (GAC)[5], and correctly registered in the cryptographic configuration of .NET. While this is definitely the best way to go using version 3.5 of the .NET framework, it adds a certain complexity to the deployment.

An interesting side note can be made on the *HashName* property. The fact that this property has a public setter, can lead to inconsistencies, as shown by listing 10, where the output of a *HMACSHA1* is in fact an MD5 HMAC.

```
HMACSHA1 hmacSHA1 = new HMACSHA1(key);
hmacSHA1.HashName = "MD5";
hmacSHA1.ComputeHash(message);
byte[] result = hmacSHA1.Hash;
```

Listing 10. Inconsistency induced by an incorrect using of the *HashName* property. The result of the *HMACSHA1* is in fact an MD5 HMAC.

Alternatively, the required internal fields can be set by means of reflection. Using reflection to set internal attributes of core classes of the .NET Framework is however prone to compatibility issues with future versions of the .NET Framework, and is definitely not encouraged.

A similar, though less important problem occurs due to the *InitializeKey* method being marked as internal. A deriving class wanting to set the key in the constructor is forced to use the *Key* property, as the *InitializeKey* method is internal. The *Key* property, however, is overridable. It is considered bad practice to call overridable members in constructors, as a deriving class could have its members called before being initialized[6]. For most cases, the class deriving from the *HMAC* class could be marked as sealed (*final* in Java). However, cases could be found where an additional level in the class hierarchy is desirable. An example would be an abstract *smart card HMAC* class that derives from the *HMAC* class and bundles smart card specific functionality (for instance, smart card parameter management). Specific smart card HMAC implementations could then derive from this *smart card HMAC* class.

C. Suggested solution

Using protected methods and fields is a well-established design pattern to allow for a shielded way of code sharing between base classes and their children. The *HMAC* class would benefit in extensibility with a promotion of the internal fields and methods to protected fields and methods. With protected fields, the need for a setter on the *HashName* property disappears. Removing this setter would prevent inconsistencies as illustrated in listing 10.

The change in visibility of the internal fields and methods preserves the backwards compatibility and enables the reusability of the *HMAC* class by third-party developers.

While the setter on the *HashName* property is not used in the .NET Framework itself, its removal could break the backwards compatibility with third party applications and CSPs that rely on this setter. Hence, removing the HashName setter is an optional suggestion.

V. Certificates

A. Certificates in the .NET Framework

The .NET Framework ships with an *X509Certificate* class that represents an X.509 v3 certificate. Version 2.0 of the .NET Framework extended this class with an *X509Certificate2* class that contains additional features. These

[5]The global assembly cache is a repository of shared .NET libraries

[6]rule CA2214 in Microsoft FxCop

classes are widely used in the .NET Framework itself, e.g. for secure network channels, encrypted and signed XML, permissions, policies, Cryptographic Message Syntax (CMS), ...

An interesting feature added by the *X509Certificate2* class is the possibility to set an *AsymmetricAlgorithm* instance as the private key for the certificate. This feature is of particular interest when using smart cards to perform asymmetric cryptographic operations. One of the advantages of using smart cards for this purpose is that the private key never has to leave the smart card. The data to sign/decrypt can be passed on to the smart card, which will unlock the private key for internal use, perform the operations, and return the result. This means that, although the private key will never be available to the outside world, it is possible to define an *AsymmetricAlgorithm* class to issue the commands to the smart card.

Consider the code in listing 11 to generate a digital (PKCS #7) signature. The *SignedCms* class does not accept any *AsymmetricAlgorithm* parameters to compute the digital signature. The X.509 certificate of the signer can, however, be passed to the *SignedCms* class through the *CmsSigner* class. This will allow the *SignedCms* class to compute a digital signature using the private key information from the certificate.

```
ContentInfo contentInfo =
    new ContentInfo(myMessage);
SignedCms signedCms =
    new SignedCms(contentInfo);
CmsSigner cmsSigner =
    new CmsSigner(mySignerCert);
signedCms.ComputeSignature(cmsSigner);
```

Listing 11. Sample code to generate a PKCS #7 signature.

To generate a PKCS #7 signature through the smart card RSA cryptographic service provider introduced in section I, the previous paragraphs suggest that the *PrivateKey* property of the signer certificate could be set to a smart card RSA provider, which would allow the code in listing 11 to be used without modifications. In practice, however, this is not the case.

B. Identified Problem

Although the *PrivateKey* property of the *X509Certificate2* class expects an *AsymmetricAlgorithm* instance, the setter immediately casts the *AsymmetricAlgorithm* parameter to the *ICspAsymmetricAlgorithm* interface. This interface, implemented by the *DSACryptoServiceProvider* and the *RSACryptoServiceProvider* classes, allows for interaction with the CryptoAPI. This is problematic, because only wrappers around the CryptoAPI can offer this kind of interaction.

Because our smart card based RSA implementation does not rely on the CryptoAPI, it can not implement the *ICspAsymmetricAlgorithm* interface. Consequently, the *PrivateKey* property of the *X509Certificate* class will not accept an instance of our RSA implementation. This implies that a complete reimplementation of the CMS functionality is required in order to generate *PKCS #7* signatures with custom .NET CSPs deriving from the *AsymmetricAlgorithm* class.

C. Suggested solutions

As the use of X.509 certificates throughout the .NET Framework is strongly connected to the native CryptoAPI, no easy solution can be found for this problem. The fact that the parameter for the *PrivateKey* property of the *X509Certificate2* class is of the *AsymmetricAlgorithm* type is positive, because this opens opportunities for a more loose coupling between the .NET cryptographic framework and the native CryptoAPI in the future, whereas a parameter of type *ICspAsymmetricAlgorithm* would have limited the possibilities. However, allowing to get and set *AsymmetricAlgorithm* instances that do not implement the *ICspAsymmetricAlgorithm* for this property would be beneficial.

As the required changes can be performed internally, and no class interface has to be changed, full backwards compatibility would be preserved.

VI. Conclusion

One of the design goals of the .NET cryptographic framework is to provide an extensible platform for developers, that offers algorithmic and implementation independency. While it achieves these goals up to a certain level, in practice a number of problems occur. These problems get bigger when trying to deal with less conventional implementations of cryptographic algorithms, like cryptographic operations on smart cards.

This paper identified a number of problems, and proposed solutions to alleviate these issues. A common goal of all solutions is to maintain backwards compatibility, and minimize the impact of the changes on the existing cryptographic framework, while improving the implementation independency of the API.

Some of the problems mentioned, like the problems with *asymmetric algorithms* (section II), *symmetric algorithms* (section III) and *keyed hash message authentication codes* (section IV) are relatively easy to solve, effectively requiring only minor changes to the existing cryptographic library, localized in a minimal set of classes, and preserving backwards compatibility.

Other problems, like the problems with the X.509 certificates (section V) are more deeply rooted. While the presented suggestions preserve backwards compatibility, they would have a higher change impact on the inner workings of the cryptographic framework. They could, however, be kept in mind during the further development of the .NET Framework, as we believe they would improve the implementation independency of the cryptographic API.

References

[1] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. *Advances in Cryptology - CRYPTO '98*, 1462/1998:629–660, 1998.

[2] Jean-Sébastien Coron, Marc Joye, David Naccache, and Pascal Paillier. New attacks on PKCS#1 v1.5 encryption. *Advances in Cryptology - Eurocrypt 2000*, 1807:369–379, 2000.

[3] Richard J. Lipton Dan Boneh, Richard A. DeMillo. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology*, 14:101–119, 2001.

[4] Bart Preneel Danny De Cock, Karel Wouters. Introduction to the belgian eid card. *Public Key Infrastructure*, 3093/2004:1–13, 2004.

[5] David M'Raihi David Naccache. Cryptographic smart cards. *IEEE Micro*, 16/3:14–24, 1996.

[6] Mark Looi Gary Gaskell. Integrating smart cards into authentication systems. *Cryptography: Policy and Algorithms*, 1029/1996:270–281, 1996.

[7] Amir Herzberg. Payments and banking with mobile personal devices. *Wireless networking security*, 46/5:53–58, 2003.

[8] ISO/IEC. *ISO/IEC 7816-8 Identification cards - Integrated circuit cards - Commands for security operations*, second edition, 2004.

[9] ISO/IEC. *ISO/IEC 7816-4 Identification cards - Integrated circuit cards - Organization, security and commands for interchange*, second edition, 2005.

[10] David Wagner John Kelsey, Bruce Schneier. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. *Information and Communications Security: First International Conference*, pages 233–246, 1997.

[11] Amit Klein. Openbsd dns cache poisoning and multiple o/s predictable ip id vulnerability. 2007.

[12] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, 1109:104–113, 1996.

[13] Hongbo Yu Xiaoyun Wang, Yiqun Lisa Yin. Finding collisions in the full sha-1. *Advances in Cryptology - CRYPTO 2005*, 3621/2005:17–36, 2005.

[14] Xuejia Lai Hongbo Yu Xiaoyun Wang, Dengguo Feng. Collisions for hash functions md4, md5, haval-128 and ripemd. *Cryptology ePrint Archive*, 2004/199:2004.