TMS Cryptography Pack
DEVELOPERS GUIDE

v4.3.0, November 2020

Web: https://www.tmssoftware.com
Email : info@tmssoftware.com

Contents

Availability

TMS Cryptography Pack is available as VCL and FMX component set for Delphi and C++Builder.

TMS Cryptography Pack is available for Delphi XE2, XE3, XE4, XE5, XE6, XE7, XE8, 10 Seattle, 10.1 Berlin, 10.2 Tokyo, 10.3 Rio, 10.4 Sydney & C++Builder XE2, XE3, XE4, XE5, XE6, XE7, XE8, 10 Seattle, 10.1 Berlin, 10.2 Tokyo, 10.3 Rio, 10.4 Sydney.

TMS Cryptography Pack has been designed for and tested with: Windows Vista, Windows 7, Windows 8, Windows 10, OSX 10.12.2 and iOS 10.2 or newer.

TMS Cryptography Pack supports following targets: Win32, Win64, Android32, Android64, OSX32, OSX64, iOS32, iOS64, (Linux with no guarantees).

If you want to use TMS Cryptography Pack in Win64 target, there are two options:
- If you have Tokyo 10.2.1 or newer, you must uncomment {$define IDEVERSION1021} in tmscrypto.inc file;
- If you have older version than 10.2.1, you must copy RandomDLL.dll from the Win64 directory to C:\Windows\System32 if you are running 64-bit Windows or to C:\Windows\SysWOW64 if you are running 32-bit Windows.

For the use of TMS Cryptography Pack on iOS, Android or Linux, you must add the directory “libAndroid” or “libAndroid64” or “libiOSDevice32” or “libiOSDevice64” or “libLinux” in the Search Path of the Project options.

Online references

TMS software website:
https://www.tmssoftware.com

TMS Cryptography Pack page:
https://www.tmssoftware.com/site/tmscrypto.asp

TMS Cryptography Pack is available separately and also as part of:

-TMS ALL-ACCESS: https://www.tmssoftware.com/site/tmsallaccess.asp

Description

TMS Cryptography Pack is a software library that provides various algorithms used to encrypt, sign and hash data. This library has been developed by Cyberens.

This manual provides a complete description of how to use the library and its various features. Each section corresponds to an algorithm used in cryptography and a class into TMS Cryptography Pack.
The different algorithms are the following:

- ✓ AES (modes ECB-CBC-OFB-CTR)
- ✓ AES MAC
- ✓ AES GCM
- ✓ SPECK
- ✓ RSA
- ✓ ECDSA and EdDSA
- ✓ ECIES
- ✓ SALSA
- ✓ SHA-2
- ✓ SHA-3
- ✓ PBKDF 2
- ✓ HKDF
- ✓ Blake2B
- ✓ RIPEMD-160
- ✓ Argon2
- ✓ Generation of X509 self-signed certificates
- ✓ Generation of X509 CSR
- ✓ XAdES
- ✓ CAdES
- ✓ PAdES
- ✓ TLSH

AES (modes ECB-CBC-OFB-CTR)

AES or Advanced Encryption Standard is a symmetric encryption algorithm. It has become a standard since 2002 in USA, described in the FIPS PUB 197. Its input is a 128-bit message and its output is a 128-bit cipher text. Depending on the version, the key length is 128 bits, 192 bits or 256 bits. To encrypt messages of different lengths, we use different encryption modes:

- ✓ ECB (Electronic Code Book): it is the simplest mode. The message to encrypt is divided into blocks of 128 bits and each block is encrypted separately with the same key.
- ✓ CBC (Cipher Block Chaining): it XORs the 128-bit first block of clear text with a 128-bit initialisation vector. Then it encrypts the result with AES. For each new block, it uses the previous cipher text as the initialisation vector.
- ✓ OFB (Output Feedback): an initialisation vector is encrypted with AES, then XORed with the first block of clear text, to obtain the first block of cipher text. Then this encrypted initialisation vector is reused as the initialisation vector for the next block.
- ✓ CTR (Counter): it encrypts a counter, which is incremented for each block. Then each counter is XORed with a block of clear text to obtain a block of cipher text.

These modes are described in the NIST Special Publication 800-38A.
The AES class is:

```
TAESKeyLength = (kl128, kl192, kl256);
TAESType = (atECB, atCBC, atOFB, atCTR);
TIVMode = (rand, userdefined);
TPaddingMode = (PKCS7, nopadding);

TAESEncryption = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TAESKeyLength; key: string; AType: TAESType;
      paddingMode: TPaddingMode; outputFormat: TConvertType;
      uni: TUnicode); overload;
    Constructor Create(keyLength: TAESKeyLength; key: string; AType: TAESType;
      paddingMode: TPaddingMode; outputFormat: TConvertType; uni: TUnicode;
      IV: string); overload;
    Destructor Destroy; override;
    function Encrypt(s: string): string;
    function Decrypt(s: string): string; overload;
    function Decrypt(s: string, var o: string): Integer; overload;
    procedure EncryptFileW(s, o: string);
    function DecryptFileW(s, o: string): Integer;
    procedure EncryptStream(s: TStream; var o: TStream);
    function DecryptStream(s: TStream; var o: TStream): Integer;
  published
    property key: string read FKey write SetKey;
    property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
      kl128;
    property AType: TAESType read FType write FType default atcbc;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property IVMode: TIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property paddingMode: TPaddingMode read FPaddingMode write FPaddingMode
      default PKCS7;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
```

```
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors and the destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(keyLength: TAESKeyLength; key: string; AType: TAESType; paddingMode: TPaddingMode; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor with IVMode = rand
- `Constructor Create(keyLength: TAESKeyLength; key: string; AType: TAESType; paddingMode: TPaddingMode; outputFormat: TConvertType; uni: TUnicode; IV: string); overload;` the constructor with IVMode = userdefined
- `Destructor Destroy; override;` to zero the key

The public methods are:

- `function Encrypt(s: string): string;` to encrypt a string s
- `function Decrypt(s: string): string;` to decrypt a string s
- `function Decrypt(s: string; var o: string): Integer;` to decrypt a string s and return 0 if success and error code if failure
- `procedure EncryptFileW(s, o: string);` to encrypt a file whose path is s and the encrypted file path is o
- **function** DecryptFileW(s, o: string); to decrypt a file whose path is s and the decrypted file path is o and return 0 if success and error code if failure
- `procedure EncryptStream(s: TStream; var o: TStream);` to encrypt the stream s into the stream o
- `function DecryptStream(s: TStream; var o: TStream);` to decrypt the stream s into the stream o and return 0 if success and error code if failure

The properties are:

- `property Key: string read FKey write SetKey;` to read and write the key
- `property KeyLength: TAESKeyLength read FKeyLength write SetKeyLength;` to read and write the key length in bits (128, 192 or 256 bits)
- `property AType: TAESType read FType write FType;` to read and write the encryption mode (ECB, CBC, OFB or CTR)
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property IVMode: TIVMode read FIVMode write FIVMode;` to read and write the IV mode, userdefined or rand.
- `property IV: string read FIV write SetIV;` to read and write the IV of 16 bytes if the IV mode is userdefined (in rand mode, the IV is randomly generated and added to the encrypted text)
- `property PaddingMode: TPaddingMode read FPaddingMode write FpaddingMode;` to read and write the padding mode, PKCS7 or nopadding. In PKCS7, the length of the encrypted text is always the length of the clear text + 16 bytes (plus 16 bytes in the case of rand IV mode). In nopadding mode, the length of the clear text must be a multiple of 16 bytes, and no padding is added to the clear text.
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the input file name has Unicode characters

- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during encryption / decryption of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of encryption with AES

```
var
  aes: TAESEncryption;
  cipher: string;
begin
  aes:= TAESEncryption.Create;
  aes.AType:= atCBC;
  aes.KeyLength:= kl256;
  aes.Unicode := yesUni;
  aes.Key:= '12345678901234567890123456789012';
  aes.OutputFormat:=hexa;
  aes.PaddingMode:= TpaddingMode.PKCS7;
  aes.IVMode:= TIVMode.rand;
  cipher:= aes.Encrypt('test');
  aes.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

AES MAC

To produce a message authentication code (MAC), we use the MAC mode. It is described in the NIST Special Publication 800-38B.
The AES MAC class is:

```
TAESKeyLength = (kl128,kl192,kl256);
TAESMAC = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
      tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode);
      overload;
    Destructor Destroy; override;
    function Generate(s: string): string;
    function Verify(s, t: string): integer;
    function GenerateFromFile(s: string): string;
    function VerifyFromFile(s, t: string): integer;
    function GenerateFromStream(s: TStream): string;
    function VerifyFromStream(s: TStream; t: string): integer;
  published
    property key: string read FKey write SetKey;
    property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
      kl128;
    property tagSizeBits: integer read FTagSizeBits write SetTagSizeBits default
      128;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors and the destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(keyLength: TAESKeyLength; key: string; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to set all the properties
- `Destructor Destroy; override;` to zero the key

The public methods are:

- `function Generate(s: string): string;` to generate a tag from a string s
- `function Verify(s, t: string): Integer;` to verify the tag t from the string s
- `function GenerateFromFile(s: string): string;` to generate a tag from a file whose path is s
- `function VerifyFromFile(s, t: string): Integer;` to verify a tag t from a file whose path is s
- `function GenerateFromStream(s: TStream): string;` to generate a tag from the stream s

- `function VerifyFromStream(s: TStream; t: string): integer;` to verify the tag t from the stream s

The properties are:

- `property Key: string read FKey write SetKey;` to read and write the key
- `property KeyLength: TAESKeyLength read FKeyLength write SetKeyLength;` to read and write the key length in bits (128, 192 or 256 bits)
- `property TagSizeBits: Integer read FTagSizeBits write SetTagSizeBits;` to read and write the tag length in bits (<= 128 bits)
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the input file name has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during generation / verification of the tag of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to generate a tag from a string with AES MAC

```
var
  aesmac: TAESMAC;
  tag: string;
begin
  aesmac:= TAESMAC.Create;
  aesmac.KeyLength:= kl256;
  aesmac.Key:= '12345678901234567890123456789012';
  aesmac.TagSizeBits:= 128;
  aesmac.OutputFormat:= hexa;
  aesmac.Unicode:= noUni;
  tag:= aesmac.Generate('test');
  aesmac.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

AES GCM

The last mode is the Galois Counter Mode, that encrypts the message using the CTR mode and products a tag using a hash function. It is described in the NIST Special Publication 800-38D. This mode allows the user to verify the integrity of some additional data, without encrypt it.
The AES-GCM class is:

```
TAESKeyLength = (kl128, kl192, kl256);
TIVMode = (rand, userdefined);

TAESGCM = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
      tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode);
      overload;
    Constructor Create(keyLength: TAESKeyLength; key: string;
      tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode;
      IVLength: integer; IV: string); overload;
    Destructor Destroy; override;
    function EncryptAndGenerate(s, a: string): string;
    function DecryptAndVerify(s, a: string; var o: string): integer;
    procedure EncryptAndGenerateFromFile(inputPath, outputPath, addDataPath,
      tagPath: string);
    function DecryptAndVerifyFromFile(inputPath, outputPath, addDataPath,
      tagPath: string): integer;
    procedure EncryptAndGenerateFromStream(inputStream: TStream;
      var outputStream: TStream; addDataStream: TStream;
      var tagStream: TStream);
    function DecryptAndVerifyFromStream(inputStream: TStream;
      var outputStream: TStream; addDataStream: TStream;
      var tagStream: TStream): integer;
  published
    property key: string read FKey write SetKey;
    property keyLength: TAESKeyLength read FKeyLength write SetKeyLength default
      kl128;
    property tagSizeBits: integer read FTagSizeBits write SetTagSizeBits default
      128;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property IVMode: TIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property IVLength: integer read FIVLength write SetIVLength;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
    property UseOldGCM: boolean read FUseOldGCM write FUseOldGCM default false;
  end;
```

The constructors and the destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor

- `Constructor Create(keyLength: TAESKeyLength; key: string; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor with IVMode = rand
- `Constructor Create(keyLength: TAESKeyLength; key: string; tagSizeBits: integer; outputFormat: TConvertType; uni: TUnicode; IVLength: integer; IV: string); overload;` the constructor with IVMode = userdefined
- `Destructor Destroy; override;` to zero the key

The methods are:

- `function EncryptAndGenerate(s, a: string): string;` to encrypt the string s and generate a tag from s and additional data a (the output string and the tag are concatenated)
- `function DecryptAndVerify(s, a: string; var o: string): integer;` to decrypt the string s and verify the tag (contained in s) associated with s and the additional data a, the resulting string is the o string. This function returns 0 if the decryption has succeeded and an error code if it has failed.
- `procedure EncryptAndGenerateFromFile(inputPath, outputPath, addDataPath, tagPath: string);` to encrypt a file whose path is inputPath into a file whose path is outputPath and generate a tag (associated with the inputPath file and the addDataPath file) into the file tagPath
- `function DecryptAndVerifyFromFile(inputPath, outputPath, addDataPath, tagPath: string): Integer;` to decrypt a file whose path is inputPath into a file which path is outputPath and verify the tag, associated with the outputPath file and the addDataPath file, whose path is tagPath.
- `procedure EncryptAndGenerateFromStream(inputStream: TStream; var outputStream: TStream; addDataStream: TStream; var tagStream: TStream);` to encrypt the stream inputStream into outputStream and generate a tag (associated with inputStream and addDataStream)
- `function DecryptAndVerifyFromStream(inputStream: TStream; var outputStream: TStream; addDataStream: TStream; var tagStream: TStream): integer;` to decrypt the stream inputStream into outputStream and verify the tag tagStream associated with outputStream and addDataStream

The properties are:

- `property Key: string read FKey write SetKey;` to read and write the key
- `property KeyLength: TAESKeyLength read FKeyLength write SetKeyLength;` to read and write the key length in bits (128, 192 or 256 bits)
- `property TagSizeBits: Integer read FTagSizeBits write SetTagSizeBits;` to read and write the tag length in bits (<= 128 bits)
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property IVMode: TIVMode read FIVMode write FIVMode;` to read and write the IV mode, userdefined or rand.
- `property IV: string read FIV write SetIV;` to read and write the IV of IVLength bytes if the IV mode is userdefined (in rand mode, the IV (12 bytes) is randomly generated and added to the encrypted text)
- `property IVLength: integer read FIVLength write SetIVLength;` to read and write the IV length in bytes if the IVMode is userdefined

- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during encryption / decryption of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes
- `property UseOldGCM: boolean read FUseOldGCM write FUseOldGCM default false;` to use the old version of AES GCM before 3.5 version of the library

Example of how to encrypt with AES-GCM

```
var
  aesgcm: TAESGCM;
  cipher: string;
begin
  aesgcm:= TAESGCM.Create;
  aesgcm.TagSizeBits:= 128;
  aesgcm.KeyLength:= kl256;
  aesgcm.Key:= '12345678901234567890123456789012';
  aesgcm.OutputFormat:= base64;
  aesgcm.IVMode:= TIVMode.rand;
  aesgcm.Unicode := yesUni;
  cipher:= aesgcm.EncryptAndGenerate('test', '');
  aesgcm.Free;
end;
```

All AES functions/procedures are located in the AESObj file.

RSA

RSA is an asymmetric encryption algorithm and a signature algorithm, described in 1977 by Ronald Rivest, Adi Shamir et Leonard Adleman.
To encrypt some data with RSA, it is necessary to use the public key of the recipient and to decrypt, it is necessary to use the recipient's private key.
To sign some data with RSA, it is necessary to use the sender's private key, and the recipient verifies the signature with the public key of the sender.
The RSA algorithm is not secured in its initial form. To make it secured, there are two options. The first is to use OAEP for encryption and PSS for signature. OAEP has been developed as a padding scheme. Similarly, for the signature, the secure form is called PSS. The second is to use PKCS v1.5 padding scheme for encryption and signature. OAEP, PSS and v1.5 are described in the PKCS#1 v2.2.
We will use in our algorithms, RSA keys of length 2048, 3072 or 4096 bits.
Our RSA algorithm supports sha256, sha384 and sh512 as hash functions. Sha1 will be only use for X509 certificate verification in a forthcoming release.
The RSA class is:

```
TRSAKeyLength = (kl2048, kl3072, kl4096);
TRSAEncType = (oaep, epkcs1_5);
TRSASignType = (pss, spkcs1_5);
TRSAHashfunction = (sha1, sha256, sha384, sha512);

TRSAEncSign = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; reintroduce; overload;
    Constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; hashF: TRSAHashFunction;
      outputFormat: TConvertType; uni: TUnicode); reintroduce; overload;
    Constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; hashF: TRSAHashFunction;
      outputFormat: TConvertType; uni: TUnicode; encT: TRSAEncType);
      reintroduce; overload;
    Constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; hashF: TRSAHashFunction;
      outputFormat: TConvertType; uni: TUnicode; signT: TRSASignType);
      reintroduce; overload;
    constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; privateExp: string; hashF: TRSAHashFunction;
      outputFormat: TConvertType; uni: TUnicode; CT: Boolean);
      reintroduce; overload;
    constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; privateExp: string; hashF: TRSAHashFunction;
      outputFormat: TConvertType; uni: TUnicode;
      encT: TRSAEncType; CT: Boolean); reintroduce; overload;
    constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; privateExp: string; hashF: TRSAHashFunction;
      outputFormat: TConvertType; uni: TUnicode;
      signT: TRSASignType; CT: Boolean); reintroduce; overload;
    Destructor Destroy; override;
    procedure GenerateKeys;
    procedure GenerateKeysX509Compatible(var dp, dq, p, q, inverseQ: string);
    function Encrypt(m: string): string;
    function Decrypt(m: string): string;
    function Sign(m: string): string;
    function Verify(m: string; s: string): Integer;
    function SignFile(filePath: string): string;
```

```
    function VerifySignatureFile(filePath, s: string): Integer;
    procedure FromOpenSSLCert(filePath: string);
    procedure FromOpenSSLPublicKey(filePath: string);
    procedure FromOpenSSLPrivateKey(filePath: string);
    procedure FromOpenSSLEncPrivateKey(filePath, Password: string);
    procedure FromOpenSSLCertString(cert: string);
    procedure FromOpenSSLPublicKeyString (key: string);
    procedure FromOpenSSLPrivateKeyString (key: string);
    function OpenSSLTypeFile(filePath: string): TOpenSSLFileType;
    procedure FromCertificate(CertStr: string);
    procedure FromCertificateFile(CertFile: string);
    procedure FromPrivateKey(KeyStr: string);
    procedure FromPrivateKeyFile(KeyFile: string);
    procedure FromPublicKey(KeyStr: string);
    procedure FromPublicKeyFile(KeyFile: string);

  published
    property modulus: string read FModulus write SetModulus;
    property PublicExponent: string read FPublicExponent
      write SetPublicExponent;
    property PrivateExponent: string read FPrivateExponent
      write SetPrivateExponent;
    property keyLength: TRSAKeyLength read FKeyLength write SetKeyLength
      default kl2048;
    property hashFunction: TRSAHashFunction read FHashFunction write
      FHashFunction default sha256;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property passwd: string read Fpw;
    property withOpenSSL: Boolean read FwithOpenSSL write SetwithOpenSSL
      default false;
    property constantTime: Boolean read FConstantTime write FConstantTime
      default true;
    property signType: TRSASignType read FSignType write FSignType default pss;
    property encType: TRSAEncType read FEncType write FEncType default oaep;
  end;
```

The constructors and the destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(keyLength: TRSAKeyLength; modulus: string; publicExp: string; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to instantiate a public key
- `Constructor Create(keyLength: TRSAKeyLength; modulus: string; publicExp: string; privateExp: string; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to instantiate a key pair
- `Destructor Destroy; override;` to zero the keys

The public methods are:

- `procedure GenerateKeys;` to generate the modulus, the public exponent and the private exponent

- `procedure GenerateKeysX509Compatible(var dp, dq, p, q, inverseQ: string);` to generate the modulus, the public exponent (fixed value 65537) and the private exponent, and also the private variables dp, dq, p, q and inverse (for interoperability with other libraries)
- `function Encrypt(m: string): string;` to encrypt the string m
- `function Decrypt(m: string): string;` to decrypt the string m
- `function Sign(m: string): string;` to sign the string m
- `function Verify(m: string; s: string): Integer;` to verify the signature s of the string m
- `function SignFile(filePath: string): string;` to sign a file
- `function VerifySignatureFile(filePath, s: string): Integer;` to verify the signature s of a file
- `procedure FromOpenSSLCert(filePath: string);` to import a public key from an OpenSSL Certificate (PEM format)
- `procedure FromOpenSSLPublicKey(filePath: string);` to import a public key from an OpenSSL public key (PEM format)
- `procedure FromOpenSSLPrivateKey(filePath: string);` to import a key pair from an OpenSSL private key (PEM format)
- `procedure FromOpenSSLEncPrivateKey(filePath, Password: string);` to import a key pair from an encrypted OpenSSL private key (PEM format)
- `procedure FromOpenSSLCertString(cert string);` to import a public key from the string contained in an OpenSSL Certificate (PEM format)
- `procedure FromOpenSSLPublicKeyString (key: string);` to import a public key from the string contained in an OpenSSL public key (PEM format)
- `procedure FromOpenSSLPrivateKeyString (key: string);` to import a key pair from the string contained in an OpenSSL private key (PEM format)
- `function OpenSSLTypeFile(filePath: string): TOpenSSLFileType;` to give the type an OpenSSL file
- `procedure FromCertificate(CertStr: string);` import a public key from a base64 certificate string
- `procedure FromCertificateFile(CertFile: string);` import a public key from a PEM certificate file
- `procedure FromPrivateKey(KeyStr: string);` import a private key from a base64 private key string
- `procedure FromPrivateKeyFile(KeyFile: string);` import a private key from a PEM private key file
- `procedure FromPublicKey(KeyStr: string);` import a public key from a base64 public key string
- `procedure FromPublicKeyFile(KeyFile: string);` import a public key from a PEM public key file

The properties are:

- `property Modulus: string read FModulus write SetModulus;` to read and write the modulus
- `property PublicExponent: string read FPublicExponent write SetPublicExponent;` to read and write the public exponent (16 bytes)

- `property PrivateExponent: string read FPrivateExponent write SetPrivateExponent;` to read and write the private exponent
- `property KeyLength: TRSAKeyLength read FKeyLength write SetKeyLength;` to read and write the key length in bits (2048, 3072 or 4096 bits)
- `property hashFunction: TRSAHashFunction read FHashFunction write FHashFunction;` to choose the hash function (sha256, sha384 or sha512) used to hash in the different algorithms
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters
- `property passwd: string read Fpw;` to save temporary the password to decrypt an OpenSSL encrypted private key
- `property withOpenSSL: boolean read FwithOpenSSL write SetwithOpenSSL;` to indicate whether OpenSSL is used
- `property ConstantTime: boolean read FconstantTime write FConstantTime;` to indicate whether the encryption/decryption/signature/verification uses constant time implementation.
- `property signType: TRSASignType read FSignType write FSignType default pss;` to indicate whether the signature function uses PSS or PKCS v1.5 mode.
- `property encType: TRSAEncType read FEncType write FEncType default oaep;` to indicate whether the encryption function uses OAEP of PKCS v1.5 mode

Example of how to encrypt with RSA

```
var
  rsa: TRSAEncSign;
  cipher: string;
begin
  rsa:= TRSAEncSign.Create;
  rsa.KeyLength:= kl2048;
  rsa.OutputFormat:= base64;
  rsa.GenerateKeys;
  rsa.Unicode := noUni;
  rsa.encType := oaep;
  cipher:= rsa.Encrypt('test');
  rsa.Free;
end;
```

Example of how to sign with RSA

```
var
  rsa: TRSAEncSign;
  signature: string;
begin
  rsa:= TRSAEncSign.Create;
  rsa.KeyLength:= kl2048;
  rsa.OutputFormat:= base64;
  rsa.GenerateKeys;
  rsa.Unicode := yesUni;
```

```
  rsa.hashFunction := sha256;
  rsa.signType := pss;
  signature:= rsa.Sign('test');
  rsa.Free;
end;
```

All RSA functions/procedures are located in the RSAObj file.

ECDSA, EdDSA, ECDH and ECIES

EdDSA is a digital signature algorithm using Edwards elliptic curves. It has been developed by a team directed by Daniel J. Bernstein. Three algorithms are implemented in this library, EdDSA25519, whose public key is encoded on 256 bits, Ed448, whose public key is encoded in 456 bits and EdDSA511187, whose public key is encoded on 512 bits.
Elliptic Curve Integrated Encryption Scheme (ECIES) is an asymmetric encryption scheme using elliptic curves. In this library, we have used Edwards curves, Curve25519 and Curve511187 because they ensure a good security level and allow us to reuse a part of the algorithms already implemented for EdDSA25119 and EdDSA511187.
ECDSA is a digital signature algorithm using elliptic curves cryptography. NIST standardized three curves, P-256, P-384 and P-521 in July 2013 in FIPS 186-4.
ECDH is an anonymous key agreement protocol that allows two parties, each having an elliptic-curve public–private key pair, to establish a shared secret over an insecure channel. This shared secret may be directly used as a key, or to derive another key. The key, or the derived key, can then be used to encrypt subsequent communications using a symmetric-key cipher. It is a variant of the Diffie–Hellman protocol using elliptic-curve cryptography. ECDH is only implemented for the curve p-256, p-384 and p-521.
The ECC (Elliptic Curve Cryptography) class is:

```
TECCType = (cc25519, cc448, cc511187, p256, p384, p521);
TNaCl = (naclno, naclyes);

TECCEncSign = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(AType: TECCType; PublicKey: string; NaCl: TNaCl;
      outputFormat: TConvertType; uni: TUnicode); overload;
    Constructor Create(AType: TECCType; PublicKey: string; PrivateKey: string;
      NaCl: TNaCl; outputFormat: TConvertType; uni: TUnicode); overload;
    Destructor Destroy; override;
    procedure GenerateKeys;
    function Encrypt(m: string): string;
    function Decrypt(m: string): string;
    function Sign(m: string): string;
    function Verify(m: string; s: string): integer;
    function SignFile(filePath: string): string;
    function VerifySignatureFile(filePath, s: string): Integer;
    function GenerateSharedSecret(PeerPublicKey: string): string;
    procedure FromCertificate(CertStr: string);
    procedure FromCertificateFile(CertFile: string);
    procedure FromPrivateKey(KeyStr: string);
    procedure FromPrivateKeyFile(KeyFile: string);
    function FromPublicKey(KeyStr: string): string;
    function FromPublicKeyFile(KeyFile: string): string;
  published
    property PublicKey: string read FPublicKey write SetPublicKey;
    property PrivateKey: string read FPrivateKey write SetPrivateKey;
    property ECCType: TECCType read FECCType write SetType default cc25519;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property NaCl: TNaCl read FNaCl write FNaCl default NaClno;
    property Unicode: TUnicode read FUni write FUni default yesUni;
  end;
```

The constructors and destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(AType: TECCType; PublicKey: string; NaCl: TNaCl; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to instantiate a public key
- `Constructor Create(AType: TECCType; PublicKey: string; PrivateKey: string; NaCl: TNaCl; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to instantiate a key pair
- `Destructor Destroy; override;` to zero the keys

The public methods are:

- `procedure GenerateKeys;` to generate the public and private keys
- `function Encrypt(m: string): string;` to encrypt the string m
- `function Decrypt(m: string): string;` to decrypt the string m
- `function Sign(m: string): string;` to sign the string m
- `function Verify(m: string; s: string): Integer;` to verify the signature s of the string m
- `function SignFile(filePath: string): string;` to sign a file
- `function VerifySignatureFile(filePath, s: string): Integer;` to verify the signature s of a file
- `function GenerateSharedSecret(PeerPublicKey: string): string;` ECDH algorithm to compute a shared secret with an other entity, who gave us his/her public key. The other entity can compute the same secret key with our public key.
- `procedure FromCertificate(CertStr: string);` import a public key from a base64 certificate string
- `procedure FromCertificateFile(CertFile: string);` import a public key from a PEM certificate file
- `procedure FromPrivateKey(KeyStr: string);` import a private key from a base64 private key string
- `procedure FromPrivateKeyFile(KeyFile: string);` import a private key from a PEM private key file
- `function FromPublicKey(KeyStr: string): string;` extract a public key from a PEM public key
- `function FromPublicKeyFile(KeyFile: string): string;` extract a public key from a PEM public key file
-

The properties are:

- `property PublicKey: string read FPublicKey write SetPublicKey;` to read and write the public key
- `property PrivateKey: string read FPrivateKey write SetPrivateKey;` to read and write the private key
- `property ECCType: TECCType read FECCType write SetType;` to read and write the curve name

- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property NaCl: TNaCl read FNaCl write FNaCl;` to use an EdDSA algorithm interoperable with NaCl software library (available only for ed25519)
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters

Example of how to encrypt with ECIES

```
var
  ecc: TECCEncSign;
  cipher: string;
begin
  ecc:= TECCEncSign.Create;
  ecc.ECCType:= cc25519;
  ecc.OutputFormat:= base64;
  ecc.Unicode:= noUni;
  ecc.NaCl := naclno;
  ecc.GenerateKeys();
  cipher:= ecc.Encrypt('test');
  ecc.Free;
end;
```

Example of how to sign with EdDSA

```
var
  ecc: TECCEncSign;
  signature: string;
begin
  ecc:= TECCEncSign.Create;
  ecc.ECCType:= cc25519;
  ecc.OutputFormat:= base64;
  ecc.Unicode:= yesUni;
  ecc.NaCl := naclno;
  ecc.GenerateKeys;
  signature:= ecc.Sign('test');
  ecc.Free;
end;
```

Example of how to share a secret key with ECDH from a PEM peer public key

```
var
  ecc: TECCEncSign;
  sharedSecret: string;
begin
  ecc:= TECCEncSign.Create;
  ecc.ECCType:= p256;
  ecc.OutputFormat:= base64;
  ecc.Unicode:= yesUni;
  ecc.GenerateKeys;
  sharedSecret:= ecc.GenerateSharedSecret(ecc.FromPublicKey(PeerPublicKey));
  ecc.Free;
end;
```

All ECC functions/procedures are located in the ECCObj file.

SALSA

Salsa20 is a stream encryption algorithm proposed by Daniel Bernstein.
The SALSA class is:

```
TSalsaKeyLength = (skl128, skl256);

TSalsaEncryption = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(keyLength: TSalsaKeyLength; key: string;
      outputFormat: TConvertType; uni: TUnicode); overload;
    Destructor Destroy; override;
    function Encrypt(s: string): string;
    function Decrypt(s: string): string;
    procedure EncryptFile(s, o: string);
    procedure DecryptFile(s, o: string);
    procedure EncryptStream(s: TStream; var o: TStream);
    procedure DecryptStream(s: TStream; var o: TStream);
  published
    property key: string read FKey write SetKey;
    property keyLength: TSalsaKeyLength read FKeyLength write SetKeyLength
      default skl128;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors and destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(keyLength: TSalsaKeyLength; key: string; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to set all the parameters
- `Destructor Destroy; override;` to zero the key

The public methods are:

- `function Encrypt(s: string): string;` to encrypt the string s
- `function Decrypt(s: string): string;` to decrypt the string s
- `procedure EncryptFile(s, o: string);` to encrypt the file whose path is s in the file whose path is o
- `procedure DecryptFile(s, o: string);` to decrypt the file whose path is s in the file whose path is o
- `procedure EncryptStream(s: TStream; var o: TStream);` to encrypt the stream s in the stream o
- `procedure DecryptStream(s: TStream; var o: TStream);` to decrypt the stream s in the stream o

The properties are:

- `property Key: string read FKey write SetKey;` to read and write the key
- `property KeyLength: TSalsaKeyLength read FKeyLength write SetKeyLength;` to read and write the key length in bits (256 or 512 bits)
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the file name has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during encryption / decryption of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to encrypt with SALSA

```
var
  salsa: TSalsaEncryption;
  cipher: string;
begin
  salsa:= TSalsaEncryption.Create;
  salsa.KeyLength:= skl128;
  salsa.Key:= '0123456789012345';
  salsa.Unicode := yesUni;
  salsa.OutputFormat:= hexa;
  cipher:= salsa.Encrypt('test');
  salsa.Free;
end;
```

All SALSA functions/procedures are located in the SALSAObj file.

SHA-2

SHA-2 (Secure Hash Algorithm) is a family of hash functions that have been designed by the National Security Agency (NSA) of the USA, on the model of the now deprecated SHA-1 and SHA-0 functions. The algorithms of the SHA-2 family, SHA-224, SHA-256, SHA-384 and SHA-512 are described and published along with SHA-1 in the FIPS 180-2 (Secure Hash Standard). In this library, only SHA-256, SHA-384 and SHA-512 have been implemented.
SHA-2 is described in the FIPS PUB 180-4.
The SHA2 class is:

```
TSHA2Hash = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType;
      uni: TUnicode); overload;
    function Hash(s: string): string;
    function HashFile(s: string): string;
    function HashStream(s: TStream): string;
    function HMAC(s, k: string): string;
    function VerifyHMAC(s, k, h: string): Integer;
  published
    property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits
      default 256;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to set all the parameters

The public methods are:

- `function Hash(s: string): string;` to hash the string s
- `function HashFile(s: string): string;` to hash the file whose path is s
- `function HashStream(s: TStream): string;` to hash the stream s
- `function HMAC(s, k: string): string;` to generate a hmac from a string s and a key k
- `function VerifyHMAC(s, k, h: string): Integer;` to verify the hmac h associated with the string s and the key k

The properties are:

- `property HashSizeBits: Integer read FHashSizeBits write SetHashSizeBits;` to read and write the number of bits (256, 384 or 512) of the hash
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)

- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during hashing of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to hash with SHA-2

```
var
  sha2: TSHA2Hash;
  hash: string;
begin
  sha2:= TSHA2Hash.Create;
  sha2.HashSizeBits:= 256;
  sha2.OutputFormat:= hexa;
  sha2.Unicode:= noUni;
  hash:= sha2.Hash('test');
  sha2.Free;
end;
```

Example of how to generate a HMAC with SHA-2

```
var
  sha2: TSHA2Hash;
  hash: string;
  k: string;
begin
  sha2:= TSHA2Hash.Create;
  sha2.HashSizeBits:= 256;
  sha2.OutputFormat:= hexa;
  sha2.Unicode := yesUni;
  k:= '0123456789012345';
  hash:= sha2.HMAC('test', k);
  sha2.Free;
end;
```

All HASH functions/procedures are located in the HashObj file.

SHA-3

SHA-3 comes from the NIST hash function competition which elected the algorithm Keccak on October 2, 2012. It is not intended to replace SHA-2 but to provide an alternative following the possibilities of attacks on the deprecated standards MD5, SHA-0 and SHA-1. This library allows hashing with the SHA-3 standard algorithm but also with the SHA-3 XOF (Extendable-Output Function) algorithm which allows to have a variable length output. SHA-3 is described in the FIPS PUB 202. This library includes the SHA3 Derived functions cSHAKE, KMAC and TupleHash, described in NIST Special Publication (SP) 800-185.
cSHAKE allows a user to add a salt to the hashed output. KMAC provides pseudorandom function and keyed hash function with variable-length outputs. And TupleHash provides function that hashes tuples of input strings correctly and unambiguously.
The SHA3 class is:

```
TSHA3Type = (tsha, txof);

TSHA3Hash = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType;
      uni: TUnicode); overload;
    Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType;
      uni: TUnicode; version: Integer); overload;
    function Hash(s: string): string;
    function HashFile(s: string): string;
    function HashStream(s: TStream): string;
    function cSHAKEHash(s, salt: string): string;
    function KMACHash(k, s, salt: string): string;
    function TupleHash(s: array of string; salt: string): string;
    function HMAC(s, k: string): string;
    function VerifyHMAC(s, k, h: string): Integer;
  published
    property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits
      default 256;
    property version: Integer read FVersion write SetVersion default 256;
    property AType: TSHA3Type read FType write SetType default tsha;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor in tsha mode
- `Constructor Create(hashSizeBits: Integer; outputFormat: TConvertType; uni: TUnicode; version: Integer); overload;` the constructor in txof mode

The public methods are:

- `function Hash(s: string): string;` to hash the string s

- `function HashFile(s: string): string;` to hash the file whose path is s
- `function HashStream(s: TStream): string;` to hash the stream s
- `function cSHAKEHash(s, salt: string): string;` to hash the string s with a salt – to be used with txof mode
- `function KMACHash(k, s, salt: string): string;` to hash the string s with a salt and a key k – to be used with txof mode
- `function TupleHash(s: array of string; salt: string): string;` to hash the array of string s with a salt – to be used with txof mode
- `function HMAC(s, k: string): string;` to generate a hmac from a string s and a key k
- `function VerifyHMAC(s, k, h: string): Integer;` to verify the hmac h associated with the string s and the key k

The properties are:

- `property HashSizeBits: Integer read FHashSizeBits write SetHashSizeBits;` to read and write the number of bits (224, 256, 384 or 512 bits in classical SHA-3, any value in extended SHA-3) of the hash
- `property Version: Integer read FVersion write SetVersion;` to read and write the version (256 or 512) in case of extended type
- `property AType: TSHA3Type read FType write SetType;` to read and write the type, classical or extended.
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the file name has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during hashing of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to hash with SHA-3

```
var
  sha3: TSHA3Hash;
  hash: string;
begin
  sha3:= TSHA3Hash.Create;
  sha3.AType:= txof;
  sha3.HashSizeBits:= 1024;
  sha3.Version:= 512;
  sha3.OutputFormat:= base64;
  sha3.Unicode := yesUni;
  hash:= sha3.Hash('test');
  sha3.Free;
end;
```

Example of how to generate a HMAC with SHA-3

```
var
```

```
  sha3: TSHA3Hash;
  hash: string;
  k: string;
begin
  sha3:= TSHA3Hash.Create;
  sha3.AType:= tsha;
  sha3.HashSizeBits:= 256;
  sha3.OutputFormat:= hexa;
  sha3.Unicode := yesUni;
  k:= '0123456789012345';
  hash:= sha3.HMAC('test', k);
  sha3.Free;
end;
```

All HASH functions/procedures are located in the HashObj file.

SPECK

Speck is a family of lightweight block ciphers publicly released by the National Security Agency (NSA) in June 2013. Speck has been optimized for performance in software implementations. Speck is an add-rotate-xor (ARX) cipher.
Speck supports a variety of block and key sizes. A block is always two words, but the words may be 16, 24, 32, 48 or 64 bits in size. The corresponding key is 2, 3 or 4 words. The round function consists in two rotations, adding the right word to the left word, xoring the key into the left word, then and xoring the left word to the right word.
To encrypt a message with many blocks, we will use the following modes (like AES):

- ECB (Electronic Code Book)
- CBC (Cipher Block Chaining)
- OFB (Output Feedback)

The SPECK class is:

```
TSPECKWordSizeBits = (wsb16, wsb24, wsb32, wsb48, wsb64);
TSPECKKeySizeWords = (ksw2, ksw3, ksw4);
TSPECKType = (stECB, stCBC, stOFB);
TSPECKIVMode = (rand, userdefined);
TSPECKPaddingMode = (PKCS7, nopadding);

TSPECKEncryption = class(TTMSCryptBase)
 public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(wordSizeBits: TSPECKWordSizeBits;
      keySizeWords: TSPECKKeySizeWords; key: string; AType: TSPECKType;
      OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode;
      uni: TUnicode); overload;
    Constructor Create(wordSizeBits: TSPECKWordSizeBits;
      keySizeWords: TSPECKKeySizeWords; key: string; AType: TSPECKType;
      OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode; uni: TUnicode;
      IV: string); overload;
    Destructor Destroy; override;
    function Encrypt(s: string): string;
    function Decrypt(s: string): string;
    procedure EncryptFileW(s, o: string);
    procedure DecryptFileW(s, o: string);
    procedure EncryptStream(s: TStream; var o: TStream);
    procedure DecryptStream(s: TStream; var o: TStream);
  published
    property key: string read FKey write SetKey;
    property wordSizeBits: TSPECKWordSizeBits read FWordSizeBits
      write SetWordSizeBits default wsb32;
    property keySizeWords: TSPECKKeySizeWords read FKeySizeWords
      write SetKeySizeWords default ksw4;
    property AType: TSPECKType read FType write FType default stcbc;
    property OutputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property IVMode: TSPECKIVMode read FIVMode write FIVMode default rand;
    property IV: string read FIV write SetIV;
    property paddingMode: TSPECKPaddingMode read FPaddingMode
      write FPaddingMode default PKCS7;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
```

```
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors and destructor are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(wordSizeBits: TSPECKWordSizeBits; keySizeWords: TSPECKKeySizeWords; key: string; AType: TSPECKType; OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode; uni: TUnicode); overload;` the constructor to set the parameters with IVMode = rand
- `Constructor Create(wordSizeBits: TSPECKWordSizeBits; keySizeWords: TSPECKKeySizeWords; key: string; AType: TSPECKType; OutputFormat: TConvertType; paddingMode: TSPECKPaddingMode; uni: TUnicode; IV: string); overload;` the constructor to set the parameters with IVMode = userdefined
- `Destructor Destroy; override;` to zero the key

The public methods are:

- `function Encrypt(s: string): string;` to encrypt the string s
- `function Decrypt(s: string): string;` to decrypt the string s
- `procedure EncryptFileW(s, o: string);` to encrypt the file whose path is s and the encrypted file path is o
- `procedure DecryptFileW(s, o: string);` to decrypt the file whose path is s and the decrypted file path is o
- `procedure EncryptStream(s: TStream; var o: TStream);` to encrypt the stream s into the stream o
- `procedure DecryptStream(s: TStream; var o: TStream);` to decrypt the stream s into the stream o

The properties are:

- `property Key: string read FKey write SetKey;` to read and write the key
- `property WordSizeBits: TSPECKWordSizeBits read FWordSizeBits write SetWordSizeBits;` to read and write the length of the words in bits
- `property KeySizeWords: TSPECKKeySizeWords read FKeySizeWords write SetKeySizeWords;` to read and write the number of words in the key
- `property AType: TSPECKType read FType write FType;` to read and write the encryption mode (ECB, CBC or OFB)
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property IVMode: TSPECKIVMode read FIVMode write FIVMode;` to read and write the IV mode, userdefined or rand.
- `property IV: string read FIV write SetIV;` to read and write the IV of FwordSizeBits/4 bytes if the IV mode is userdefined (in rand mode, the IV is randomly generated and added to the encrypted text)
- `property PaddingMode: TSPECKPaddingMode read FPaddingMode write FpaddingMode;` to read and write the padding mode, PKCS7 or nopadding. In PKCS7, the length of the encrypted text is always the length of the clear text + FwordSizeBits/4 bytes (plus FwordSizeBits/4 bytes in the case of rand IV mode). In nopadding mode, the length of

the clear text must be a multiple of FwordSizeBits/4 bytes, and no padding is added to the clear text.

- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the file name has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during encryption / decryption of a stream
- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to encrypt with SPECK

```
var
  speck: TSPECKEncryption;
  cipher: string;
begin
  speck:= TSPECKEncryption.Create;
  speck.AType:= stCBC;
  speck.WordSizeBits:= wsb32;
  speck.KeySizeWords:= ksw4;
  speck.Key:= '0123456789012345';
  speck.OutputFormat:= hexa;
  speck.Unicode := noUni;
  speck.PaddingMode:= TSPECKPaddingMode.PKCS7;
  speck.IVMode:= TSPECKIVMode.rand;
  cipher:= speck.Encrypt('test');
  speck.Free;
end;
```

All SPECK functions/procedures are located in the SPECKObj file.

PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) is a key derivation function that is part of RSA Laboratories' Public-Key Cryptography Standards (PKCS) series, specifically PKCS \#5 v2.0, also published as Internet Engineering Task Force's RFC 2898. It replaces an earlier standard, PBKDF1, which could only produce derived keys up to 160 bits long.
PBKDF2 applies a pseudorandom function, such as a cryptographic hash, cipher, or HMAC, to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult and is known as key stretching.
It is described in NIST Special Publication 800-132.

Warning: to ensure compatibility between versions before 2.4.2, we have implemented a TPBKDF2KeyDerivationOLD class, which computes a wrong result for the PBKDF2 algorithm. Use this only if you are using the PBKDF2 algorithm from a version before 2.4.2 and you want to ensure compatibility. But it is deprecated from the 2.4.2 version of TMS Cryptography Pack.

The PBKDF2 class is:

```
THashFunction = (hsha2, hsha3);

TPBKDF2KeyDerivation = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(outputSizeBits: Integer; salt: string; counter: Integer;
      outputFormat: TConvertType; uni: TUnicode, hashF: THashFunction;
      hashSB: Integer); overload;
    function GenerateKey(s: string): string;
  published
    property outputSizeBits: Integer read FOutputSizeBits
      write SetOutputSizeBits default 128;
    property Salt: string read FSalt write FSalt;
    property counter: Integer read FCounter write FCounter default 10000;
    property hashFunction: THashFunction read FHashFunction write FHashFunction
      default hsha2;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits
      default 256;
    property Unicode: TUnicode read FUni write FUni default yesUni;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(outputSizeBits: Integer; salt: string; counter: Integer; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to set all the parameters

The public method is:

- `function GenerateKey(s: string): string;` to generate a key from a password s

The properties are:

- `property OutputSizeBits: Integer read FOutputSizeBits write SetOutputSizeBits;` to read and write the output length in bits
- `property Salt: string read FSalt write FSalt;` to read and write the salt
- `property Counter: Integer read FCounter write FCounter;` to read and write the number of iterations of the algorithm
- `property hashFunction: THashFunction read FHashFunction write FHashFunction;` to read and write the hash function used into PBKDF2 algorithm
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits;` to read and write the number of output bits of the hash function used in PBKDF2 algorithm
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters

Example of how to generate a key from a password with PBKDF2

```
var
  pbkdf2: TPBKDF2KeyDerivation;
  output: string;
begin
  pbkdf2:= TPBKDF2KeyDerivation.Create;
  pbkdf2.OutputSizeBits:= 1024;
  pbkdf2.Counter:= 10000;
  pbkdf2.Unicode:= yesUni;
  pbkdf2.OutputFormat:= base64;
  pbkdf2.Salt:= '0123456789012345678901234567890123456789012345678901234567890123456789';
  output:= pbkdf2.GenerateKey('test123');
  pbkdf2.Free;
end;
```

All KEY DERIVATION functions/procedures are located in the HashObj file.

HKDF

HKDF (HMAC-based Extract-and-Expand Key Derivation Function) is a simple key derivation function (KDF) based on a hash-based message authentication code (HMAC). The main approach HKDF follows is the "extract-then-expand" paradigm, where the KDF logically consists of two modules: the first stage takes the input keying material and "extracts" from it a fixed-length pseudorandom key, and then the second stage "expands" this key into several additional pseudorandom keys (the output of the KDF).

It can be used, for example, to convert shared secrets exchanged via Diffie–Hellman into key material suitable for use in encryption, integrity checking or authentication.

It is formally described in the RFC 5869.

The HKDF class is:

```
THashFunction = (hsha2, hsha3);

THKDFKeyDerivation = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(outputFormat: TConvertType; uni: TUnicode,
      hashF: THashFunction; hashSB: Integer); overload;
    function Extract(s, salt: string): string;
    function Expand(s, info: string; len: Integer): string;
  published
    property hashFunction: THashFunction read FHashFunction write FHashFunction
      default hsha2;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits
      default 256;
    property Unicode: TUnicode read FUni write FUni default yesUni;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(outputFormat: TConvertType; uni: TUnicode; hashF: THashFunction; hashSB: Integer); overload;` the constructor to set all the parameters

The public methods are:

- `function Extract(s, salt: string): string;` to generate a pseudo random key from a message s and a salt
- `function Expand(s, info: string; len: Integer): string;` to generate a key of length len from the resulting key of Extract function, s, and an info string

The properties are:

- `property hashFunction: THashFunction read FHashFunction write FHashFunction;` to read and write the hash function used into HKDF algorithm
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits;` to read and write the number of output bits of the hash function used in HKDF algorithm
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters

Example of how to generate a key from a password with HKDF

```
var
  hkdf: THKDFKeyDerivation;
  PRK, OKM: string;
begin
  hkdf:= THKDFKeyDerivation.Create;
  hkdf.Unicode:= yesUni;
  hkdf.OutputFormat:= base64;
  hkdf.hashFunction := hsha2;
  hkdf.hashSizeBits := 256;
  PRK:= hkdf.Extract('test123', '123456');
  OKM := hdkf.Expand(PRK, 'WebPush: info', 32);
  hkdf.Free;
end;
```

All KEY DERIVATION functions/procedures are located in the HashObj file.

Blake2

BLAKE2 is a cryptographic hash function faster than MD5, SHA-1, SHA-2, and SHA-3, yet is at least as secure as the latest standard SHA-3. BLAKE2 had been adopted by many projects due to its high speed, security, and simplicity. BLAKE2 is specified in RFC 7693. We have chosen to implement BLAKE2b that is optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes.

The Blake2B class is:

```
TBlake2BHash = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(hashSizeBytes: Integer; key: string;
      outputFormat: TConvertType; uni: TUnicode); overload;
    function Hash(s: string): string;
    function HashFile(s: string): string;
    function HashStream(s: TStream): string;
  published
    property hashSizeBytes: Integer read FHashSizeBytes write SetHashSizeBytes
      default 16;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property key: String read FKey write SetKey;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(hashSizeBytes: Integer; key: string; outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to set all the parameters

The public methods are:

- `function Hash(s: string): string;` to hash a string s
- `function HashFile(s: string): string;` to hash a file whose path is s
- `function HashStream(s: TStream): string;` to hash the stream s

The properties are:

- `property HashSizeBytes: Integer read FHashSizeBytes write SetHashSizeBytes;` to read and write the hash size in bytes
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Key: String read FKey write SetKey;` to read and write the optional key
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the file name has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during hashing of a stream

- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to hash a string with Blake2B

```
var
  blake2B: TBlake2BHash;
  output: String;
begin
  blake2B:= TBlake2BHash.Create;
  try
    blake2B.Key:= '';
    blake2B.HashSizeBytes:= 64;
    blake2B.OutputFormat:= hexa;
    blake2B.Unicode := yesUni;

    output:= blake2B.Hash('ABCDEFGH');

  finally
    blake2B.Free;
  end;
end;
```

All HASH functions/procedures are located in the HashObj file.

RIPEMD-160

RIPEMD (RACE Integrity Primitives Evaluation Message Digest) is a family of cryptographic hash functions developed in Leuven, Belgium, by Hans Dobbertin, Antoon Bosselaers and Bart Preneel at the COSIC research group at the Katholieke Universiteit Leuven, and first published in 1996. RIPEMD was based upon the design principles used in MD4, and is similar in performance to the more popular SHA-1 (NOTE: both MD4 and SHA-1 are deprecated).

RIPEMD-160 is an improved, 160-bit version of the original RIPEMD, and the most common version in the family.

The RIPEMD160 class is:

```
TRIPEMD160Hash = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(outputFormat: TConvertType; uni: TUnicode); overload;
    function Hash(s: string): string;
    function HashFile(s: string): string;
    function HashStream(s: TStream): string;
  published
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property Progress: Integer read FProgress write SetProgress;
    property OnChange: TNotifyEvent write FOnChange;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(outputFormat: TConvertType; uni: TUnicode); overload;` the constructor to set all the parameters

The public methods are:

- `function Hash(s: string): string;` to hash a string s
- `function HashFile(s: string): string;` to hash a file whose path is s
- `function HashStream(s: TStream): string;` to hash the stream s

The property is:

- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer has Unicode characters
- `property Progress: Integer read FProgress write SetProgress;` to indicate progress during hashing of a stream

- `property OnChange: TNotifyEvent write FOnChange;` to indicate that the progress changes

Example of how to hash a string with RIPEMD-160

```
var
  ripemd160: TRIPEMD160Hash;
  output: String;
begin
  ripemd160:= TRIPEMD160Hash.Create;
  try
    ripemd160.OutputFormat:= hexa;
    ripemd160.Unicode:= yesUni;
    output:= ripemd160.Hash('ABCDEFGH');
  finally
    ripemd160.Free;
  end;
end;
```

All HASH functions/procedures are located in the HashObj file.

Argon2

Argon2 is a key derivation function that was selected as the winner of the Password Hashing Competition (PHC) in July 2015. It was designed by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich from the University of Luxembourg. Argon2 provides two related versions:

- Argon2d maximizes resistance to GPU cracking attacks.
- Argon2i is optimized to resist side-channel attacks.

We have chosen to implement Argon2d with no parallelism.

The Argon2 class is:

```
TArgon2KeyDerivation = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(outputSizeBytes: Integer; salt: string; counter: Integer;
      outputFormat: TConvertType; memory: Integer; uni: TUnicode); overload;
    function GenerateKey(s: string): string;
  published
    property outputSizeBytes: Integer read FOutputSizeBytes
      write SetOutputSizeBytes default 16;
    property StringSalt: string read FStringSalt write SetStringSalt;
    property counter: Integer read FCounter write SetCounter default 10;
    property outputFormat: TConvertType read FOutputFormat write FOutputFormat
      default hexa;
    property memory: Integer read FMemory write SetMemory default 16;
    property Unicode: TUnicode read FUni write FUni default yesUni;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(outputSizeBytes: Integer; salt: string; counter: Integer; outputFormat: TConvertType; memory: Integer; uni: TUnicode); overload;` the constructor to set all the parameters

The public method is:

- `function GenerateKey(s: string): string;` to generate a key from a password s

The properties are:

- `property OutputSizeBits: Integer read FOutputSizeBits write SetOutputSizeBits;` to read and write the output length in bits
- `property StringSalt: string read FStringSalt write SetStringSalt;` to read and write the salt (16 bytes in string form)
- `property Counter: Integer read FCounter write SetCounter;` to read and write the number of iterations of the algorithm (minimum 1)
- `property OutputFormat: TConvertType read FOutputFormat write FoutputFormat;` to read and write the output format of the data (see Converter class section)
- `property Memory: Integer read FMemory write SetMemory;` to read and write the amount of memory you want to use in KB (minimum 8)

- `property Unicode: TUnicode read FUni write FUni;` to indicate whether the input buffer or the file name has Unicode characters

Example of how to generate a key from a password with Argon2

```
var
  argon2: TArgon2KeyDerivation;
  output: String;
begin
  argon2 := TArgon2KeyDerivation.Create;
  try
    argon2.OutputFormat := base64;
    argon2.OutputSizeBytes := 64;
    argon2.Counter := 10;
    argon2.Memory := 16;
    argon2.Unicode:= yesUni;
    argon2.StringSalt := 'ABCDEFGHIJKLMNOP';

    output := argon2.GenerateKey('toto23');
  finally
    argon2.Free;
  end;
end;
```

All KEY DERIVATION functions/procedures are located in the HashObj file.

TLSH

TLSH is an acronym for Trend Micro Locality Sensitive. The TMS Cryptographic Pack version of TLSH is a simplified version of Trend Micro's Jonathan Oliver, Chun Cheng, and Yanggui Chen paper and Git Repository:

- https://documents.trendmicro.com/assets/wp/wp-locality-sensitive-hash.pdf
- https://github.com/trendmicro/tlsh

TLSH generates a hash value which can be used for similarity comparisons. Similar objects will have similar hash values which allows for the detection of similar objects by comparing their hash values. Note that the byte stream should have enough complexity. For example, a byte stream of identical bytes will not generate a hash value.

The TLSH class is:

```
  Ttlsh = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; reintroduce; overload;
    destructor Destroy; override;

    function Checksum(k: integer): integer;
    function BucketValue(Bucket: integer): integer;
    function ModDiff(X, Y, R: integer): integer;
    function HDistance(Length: integer; T: Ttlsh): integer;

    function GetHash: string; overload;
    function GetHash(Buffer: array of byte; Size: integer): string; overload;
    function GetHash(FileName: String): string; overload;
  published
    property tlsHash: string read GetHash;
    property comment: string read FComment;
    property Lvalue: integer read getLvalue default 0;
    property Q1ratio: integer read getQ1ratio default 0;
    property Q2ratio: integer read getQ2ratio default 0;
    property TLSHversion: string read getVersion;
    property ShowVersion: boolean read FVer write FVer default False;
  end;

function FileDistance(MyFile1, MyFile2: String; LenDiff: boolean): integer;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; reintroduce; overload;` the default constructor

The public methods are:

- `function Checksum(k: integer): integer;` return the k-th value of checksum array
- `function BucketValue(Bucket: integer): integer;` return the Bucket-th value of Bucket array
- `function ModDiff(X, Y, R: integer): integer;` the minimum number of steps between X and Y on a circular queue of size R

- `function HDistance(Length: integer; T: Ttlsh): integer;` return the distance between the current TLSH object and the T one, truncated by Length elements.
- `function GetHash: string; overload;` return the Tlsh hash value
- `function GetHash(Buffer: array of byte; Size: integer): string; overload;` return the hash of Buffer with size Size.
- `function GetHash(FileName: String): string; overload;` return the hash of the file FileName.

The properties are:

- `property tlsHash: string read GetHash;` the resulting hash
- `property comment: string read FComment;` an error message explaining why the hash value is empty
- `property Lvalue: integer read getLvalue default 0;` to get LValue TLSH parameter that is a representation of the logarithm of the byte string length (modulo 256)
- `property Q1ratio: integer read getQ1ratio default 0;` Q1ratio is the rightmost part of the third byte that is constructed out of two 16 bit quantities derived from the quartiles: q1, q2 and q3
- `property Q2ratio: integer read getQ2ratio default 0;` Q2ratio is the leftmost part of the third byte that is constructed out of two 16 bit quantities derived from the quartiles: q1, q2 and q3
- `property TLSHversion: string read getVersion;` to get the TLSH version
- `property ShowVersion: boolean read FVer write FVer default False;` to set whether we put the version in the first byte of resulting hash

There is also a function outside the class:

`function FileDistance(MyFile1, MyFile2: String; LenDiff: boolean): integer;` to compute the TLSH distance between MyFile1 and MyFile2 files

Example of how to hash a file with TLSH

```
var
  tlsh: TTLSH;
  output: String;
begin
  tlsh:= TTLSH.Create;
  try
    tlsh.ShowVersion:= false;
    output:= tlsh.GetHash('myfile.txt');
  finally
    tlsh.Free;
  end;
end;
```

Example of how to compare 2 files with TLSH

```
var
  tlsh: TTLSH;
  output: Integer;
begin
  tlsh:= TTLSH.Create;
  try
    output := FileDistance('myfile1.txt', 'myfile2.txt', false);
  finally
    tlsh.Free;
  end;
end;
```

All TLSH functions/procedures are located in the Tlsh file.

Converter class

To display the output binary data of the library functions on a screen, we need to convert them in a printable format. We have chosen four formats:

- Hexadecimal format: consists in replacing each 4-bit block by a symbol in the list 0, …, 9, A, …, F.
- Base64 format: consists in replacing each 6-bit block by a symbol in the list a, …, z, A, …, Z, 0, …, 9, + and / (the symbol = is used in complement when the length of the data is not a multiple of 3 bytes).
- Base64url format: the same as Base64 with - in place of + and _ in place of /, to be compatible with URLs.
- Base32 format: consists in replacing each 5-bit block by a symbol in the list A, …, Z, 2, …, 7 (the symbol = is used in complement when the length of the data is not a multiple of 8 bytes).

We add the raw format to have an output format compatible with the input of some functions.

The class TConvert is the following:

```
TConvertType = (base64, hexa, base64url, base32, raw);
TUnicode = (noUni, yesUni);

TConvert = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; overload;
    Constructor Create(AType: TConvertType); overload;
{$IF (defined(MSWINDOWS) or defined(MACOS)) and (not defined(IOS))}
    function CharToFormat(charstring: PAnsiChar; charlen: Integer): string;
    function FormatToChar(str: string): PAnsiChar;
    function UnicodeToPAnsiChar(str: string): PAnsiChar;
    function PAnsiCharFromUnicodeLength(str: string): Integer;
    function StringToBuffer(str: string; u: boolean; var msgLen: Integer)
      : PAnsiChar;
    function StringToBufferA(str: string; u: boolean): PAnsiChar;
{$ELSE}
    function CharToFormat(charstring: PByte; charlen: Integer): string;
    function FormatToChar(str: string): PByte;
    function StringToBuffer(str: string; u: TUnicode; var msgLen: Integer)
      : PByte;
    function StringToBufferA(str: string; u: TUnicode): PByte;
{$IFEND}
    function UnicodeStringToFormat(str: string): string;
    function FormatToUnicodeString(str: string): string;
    function StringToByteArray(str: string): TArray<Byte>;
    function TestUnicode(str: string): Integer;
    function StringToUnicode(str: string): string;
    function StringToFormat(charstring: string): string;
    function FormatToString(str: string): string;
    function OutputFormatLength(charlen: Integer): Integer;
    function CharLength(charstring: string): Integer;
    function Base64ToHexa(base64String: string): string;
    function HexaToBase64(hexaString: string): string;
    function Base64ToBase64url(inString: string): string;
```

```
    function Base64urlToBase64(inString: string): string;
    function Base64urlToHexa(inString: string): string;
    function HexaToBase64url(inString: string): string;
    function Base32ToHexa(base32String: string): string;
    function HexaToBase32(hexaString: string): string;
    function Base32ToBase64url(inString: string): string;
    function Base64urlToBase32(inString: string): string;
    function Base32ToBase64(inString: string): string;
    function Base64ToBase32(inString: string): string;
    function KeyRSAOpenSSLToKeyTRSAEncSign(strKey: string): string;
    function KeyTRSAEncSignToKeyRSAOpenSSL(strKey: string): string;
    function Base58Encode(const value: uint64): string;
    function Base58Decode(const encoded: string): uint64;
    function TBytesToString(const t: TBytes): string;
    function StringToTBytes(const str: string): TBytes;
    function RandomString(len: Integer): string;
  published
    property AType: TConvertType read FType write FType default hexa;
  end;
```

The constructors are:

- `Constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `Constructor Create; overload;` the default constructor
- `Constructor Create(AType: TConvertType); overload;` the constructor to set the type

The public methods are:

- **function** `CharToFormat(charstring: PAnsiChar; charlen: Integer): string;` to convert a `PAnsiChar` (or a `PByte`) with length `charlen` to a string in the format defined by `AType`.

- **function** `FormatToChar(str: string): PansiChar;` to convert a formatted string to a `PAnsiChar` in binary format
- **function** `UnicodeToPAnsiChar(str: string): PAnsiChar;` to convert an Unicode string to a PAnsiChar with only UTF8 characters values
- **function** `PAnsiCharFromUnicodeLength(str: string): Integer;` to compute the length of the Unicode string in byte
- **function** `StringToBuffer(str: string; u: TUnicode; var msgLen: Integer) : PAnsiChar;` to convert a string to a PAnsiChar (or PByte for mobile platform) and return the length of PAnsiChar in msgLen value.
- **function** `StringToBufferA(str: string; u: TUnicode): PAnsiChar;` to convert a string to a PAnsiChar (or PByte for mobile platform).
- **function** `UnicodeStringToFormat(str: string): string;` to convert a Unicode string to a formatted string (hexa, base64, etc.)
- **function** `FormatToUnicodeString(str: string): string;` to convert a formatted string to an Unicode string

- **function** StringToByteArray(str: string): TArray<Byte>; to convert a string to a byte array
- **function** TestUnicode(str: string): Integer; to test whether a string has Unicode characters
- **function** StringToUnicode(str: string): string; to convert an ANSI string to an Unicode string
- **function** StringToFormat(charstring: string): string; to convert a raw string to a string in the format defined by AType
- **function** FormatToString(str: string): string; to convert a string in the format defined by AType to a raw string
- **function** OutputFormatLength(charlen: Integer): Integer; to compute the length of the formatted string from the length of the binary data
- **function** CharLength(charstring: string): Integer; to compute the length of the binary data from the formatted string
- **function** Base64ToHexa(base64String: string): string; to convert a string in base64 format to a string in hexadecimal format
- **function** HexaToBase64(hexaString: string): string; to convert a string in hexadecimal format to a string in base64 format
- **function** Base64ToBase64url(inString: string): string; to convert a string in base64 format to a string in base64url format
- **function** Base64urlToBase64(inString: string): string; to convert a string in base64url format to a string in base64 format
- **function** Base64urlToHexa(inString: string): string; to convert a string in base64url format to a string in hexadecimal format
- **function** HexaToBase64url(inString: string): string; to convert a string in hexadecimal format to a string in base64url format
- **function** Base32ToHexa(base32String: string): string; to convert a string in base32 format to a string in hexadecimal format
- **function** HexaToBase32(hexaString: string): string; to convert a string in hexadecimal format to a string in base32 format
- **function** Base32ToBase64url(inString: string): string; to convert a string in base32 format to a string in base64url format
- **function** Base64urlToBase32(inString: string): string; to convert a string in base64url format to a string in base32 format
- **function** Base32ToBase64(inString: string): string; to convert a string in base32 format to a string in base64 format
- **function** Base64ToBase32(inString: string): string; to convert a string in base64 format to a string in base32 format
- **function** KeyRSAOpenSSLToKeyTRSAEncSign(strKey: string): string; to convert an RSA key (the modulus, the public exponent or the private exponent) in OpenSSL format to an RSA key usable in TRSAEncSign
- **function** KeyTRSAEncSignToKeyRSAOpenSSL(strKey: string): string; to convert an RSA key (the modulus, the public exponent or the private exponent) in TRSAEncSign format to an RSA key in OpenSSL format

- **function** Base58Encode(const value: uint64): string; to convert an uint64 to a string in Base58

- `function Base58Decode(const encoded: string): uint64;` to convert a string in Base58 to the corresponding uint64

- `function TBytesToString(const t: TBytes): string;` to convert a TBytes into a string where each byte is a character
- `function StringToTBytes(const str: string): TBytes;` to convert a string of bytes into a TBytes
- `function RandomString(len: Integer): string;` to generate a random byte string of length len

The property is:

- `property AType: TConvertType read FType write Ftype;` to read and write the type of the conversion: hexa, base64, base64url, base32 or raw

Example of how to use AES with a key in TBytes form

Let b be a TBytes array of 16 bytes.

```
var
  aes: TAESEncryption;
  conv: TConvert;
  str: string;
begin
  aes:= TAESEncryption.Create;
  conv:= TConvert.Create;
  try
    aes.AType := atcbc;
    aes.KeyLength := kl128;
    aes.OutputFormat := hexa;
    aes.Key := conv.TBytesToString(b);
    aes.IVMode := TIVMode.rand;
    aes.PaddingMode := TPaddingMode.PKCS7;
    aes.Unicode := yesUni;

    str := AES.Encrypt('test');

  Finally
    aes.Free;
    conv.Free;
  end;
end;
```

All CONVERSION functions/procedures are located in the MiscObj file.

X509 certificates

X.509 is a standard that defines the format of public key certificates. X.509 certificates are used in many Internet protocols, including TLS/SSL, which is the basis for HTTPS, the secure protocol for browsing the web. They are also used in offline applications, like electronic signatures. An X.509 certificate contains a public key and an identity (a hostname, or an organization, or an individual), and is either signed by a certificate authority or self-signed. When a certificate is signed by a trusted certificate authority, or validated by other means, someone holding that certificate can rely on the public key it contains to establish secure communications with another party, or validate documents digitally signed by the corresponding private key.

In the library, you can decode an X509 certificate to display all the fields and verify the signature if the algorithm is supported. You can also generate a self-signed certificate (only on MacOS and Windows platform) and sign a Certificate Signing Request (CSR).

The demo on Windows of these functionalities is here:
https://www.tmssoftware.com/site/freetools.asp

The signature algorithms supported by TMS CP are: RSA-SHA256, RSA-SHA384, RSA-SHA512, ECDSA-SHA256 (corresponding to P-256 curve), ECDSA-SHA384 (corresponding to P-384 curve), ECDSA-SHA512 (corresponding to P-521 curve). All RSA algorithms are RSA PKCS#1 v1.5, with a key length of 2048 or 4096 bits. Moreover, RSA-SHA1 is supported for decoding.

The X509 certificate class is:

```
TX509Certificate = class(TTMSCryptBase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create; reintroduce; overload;
    constructor Create(RootCAPath: string); reintroduce; overload;
{$IF (defined(MSWINDOWS) or defined(MACOS)) and (not defined(IOS))}
    procedure GenerateSelfSigned;
{$IF defined(MSWINDOWS)}
    procedure DecodeCertFromPFX(PFXFilePath: string; Password: string;
        PathToOpenSSL: string);
    procedure DecodeCertAndKeyFromPFX(PFXFilePath: string; Password: string;
        PathToOpenSSL: string; KeyPath: string);
    procedure ExportToPFX(PFXFilePath: string; Password: string;
        PathToOpenSSL: string);
{$IFEND}
{$IFEND}
    procedure Decode;
    procedure SignCSR(CSRFilePath: string; outputFilePath: string); overload;
    function SignCSR(CSR: string): string; overload;
  published
    property KeyFilePath: string read FKeyFilePath write setKeyFilePath;
    property CrtFilePath: string read FCrtFilePath write setCrtFilePath;
    property signatureAlgorithm: TSignAlgo read FSignatureAlgorithm
      write setSignatureAlgorithm;
    property hashFunction: TX509HashFunction read FHashfunction write
SetHashfunction default sha256;
```

```
    property countryName: string read FSubjectCountryName write SetCountryName;
    property IssuerCountryName: string read FIssuerCountryName;
    property stateName: string read FSubjectStateName write FSubjectStateName;
    property IssuerStateName: string read FIssuerStateName;
    property localityName: string read FSubjectLocalityName
      write FSubjectLocalityName;
    property IssuerLocalityName: string read FIssuerLocalityName;
    property OrganizationName: string read FSubjectOrganizationName
      write FSubjectOrganizationName;
    property IssuerOrganizationName: string read FIssuerOrganizationName;
    property OrganizationUnitName: string read FSubjectOrganizationUnitName
      write FSubjectOrganizationUnitName;
    property IssuerOrganizationUnitName: string
      read FIssuerOrganizationUnitName;
    property commonName: string read FSubjectCommonName write
        FSubjectCommonName;
    property IssuerCommonName: string read FIssuerCommonName;
    property AltName1: string read FAltName1 write FAltName1;
    property AltName2: string read FAltName2 write FAltName2;
    property AltName3: string read FAltName3 write FAltName3;
    property AltName4: string read FAltName4 write FAltName4;
    property AltName5: string read FAltName5 write FAltName5;
    property AltName6: string read FAltName6 write FAltName6;
    property isCA: boolean write FIsCA;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property version: string read FVersion;
    property serialNumber: string read FSerialNumber;
    property notBefore: string read FnotBefore;
    property notAfter: string read FnotAfter;
    property EncryptionAlgorithm: string read FEncryptionAlgorithm;
    property publicKey: string read FPublicKey;
    property modulus: string read FModulus;
    property IsSignatureValid: string read FIsSignatureValid;
    property BitSizeEncryptionAlgorithm: Integer
      read FBitSizeEncryptionAlgorithm write FBitSizeEncryptionAlgorithm;
    property ecCurve: string read FECCurve;
    property RootCAPath: string read FRootCAPath write FRootCAPath;
    property CrtStr: string read FCrtStr write FCrtStr;
    property KeyStr: string read FKeyStr write FKeyStr;
    property PSSParam: string read FPSSParam;
  end;
```

The constructors are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `constructor Create; reintroduce; overload;` the default constructor
- `constructor Create(RootCAPath: string); reintroduce; overload;` the constructor to set the path to the folder containing the CA certificates

The public methods are:

- **`procedure`** `GenerateSelfSigned;` to generate a self-signed certificate (only available on desktop platforms)

- **procedure** DecodeCertFromPFX(PFXFilePath: string; Password: string; PathToOpenSSL: string); to decode a certificate from a PFX file. Need to enter the password of the PFX file and the folder path to the openssl.exe file (only available on Windows platform)

- **procedure** DecodeCertAndKeyFromPFX(PFXFilePath: string; Password: string; PathToOpenSSL: string; KeyPath: string); to decode a certificate and save the private key (in KeyPath file) from a PFX file. Need to enter the password of the PFX file and the folder path to the openssl.exe file (only available on Windows platform)

- **procedure** ExportToPFX(PFXFilePath: string; Password: string; PathToOpenSSL: string); to export the certificate and the private key in a PFX file. Need to enter the password of the PFX file and the folder path to the openssl.exe file (only available on Windows platform)

- **procedure** Decode; to decode the fields of a certificate and verify the signature
- **procedure** SignCSR(CSRFilePath: string; outputFilePath: string); to sign a CSR in PEM format in CSRFilePath and write the output in PEM format in outputFilePath.
- **function** SignCSR(CSR: string): string; sign a CSR string and return the output certificate

The properties are:

- property KeyFilePath: string read FKeyFilePath write setKeyFilePath; to set and get the path to the private key file
- property CrtFilePath: string read FCrtFilePath write setCrtFilePath; to read and write the path to the certificate file
- property signatureAlgorithm: TSignAlgo read FSignatureAlgorithm write setSignatureAlgorithm; to read and write the path to the certificate file
- property hashFunction: TX509HashFunction read FHashfunction write SetHashfunction default sha256; to read and write the hash function
- property countryName: string read FSubjectCountryName write SetCountryName; to read and write the subject country name field
- property IssuerCountryName: string read FIssuerCountryName; to read and write the issuer country name field
- property stateName: string read FSubjectStateName write FSubjectStateName; to read and write the subject state name field
- property IssuerStateName: string read FIssuerStateName; to read and write the issuer state name field
- property localityName: string read FSubjectLocalityName write FSubjectLocalityName; to read and write the subject locality name field
- property IssuerLocalityName: string read FIssuerLocalityName; to read and write the issuer locality name field
- property OrganizationName: string read FSubjectOrganizationName write FSubjectOrganizationName; to read and write the subject organization name field
- property IssuerOrganizationName: string read FIssuerOrganizationName; to read and write the issuer organization name field

- `property OrganizationUnitName: string read FSubjectOrganizationUnitName write FSubjectOrganizationUnitName;` to read and write the subject organization unit name field
- `property IssuerOrganizationUnitName: string read FIssuerOrganizationUnitName;` to read and write the issuer organization unit name field
- `property commonName: string read FSubjectCommonName write FSubjectCommonName;` to read and write the subject common name field (mandatory)
- `property IssuerCommonName: string read FIssuerCommonName;` to read and write the issuer common name field (mandatory)
- `property AltName1: string read FAltName1 write FAltName1;` to read and write the first alternative name field
- `property AltName2: string read FAltName2 write FAltName2;` to read and write the second alternative name field
- `property AltName3: string read FAltName3 write FAltName3;` to read and write the third alternative name field
- `property AltName4: string read FAltName4 write FAltName4;` to read and write the fourth alternative name field
- `property AltName5: string read FAltName5 write FAltName5;` to read and write the fifth alternative name field
- `property AltName6: string read FAltName6 write FAltName6;` to read and write the sixth alternative name field
- `property isCA: boolean write FIsCA;` to set whether the certificate is a CA one
- `property Unicode: TUnicode read FUni write FUni default yesUni;` to indicate whether the input buffer or the file name has Unicode characters
- `property version: string read FVersion;` to read the version of the certificate
- `property serialNumber: string read FSerialNumber;` to read the serial number of the certificate
- `property notBefore: string read FnotBefore;` to read the date of emission of the certificate
- `property notAfter: string read FnotAfter;` to read the date of validity of the certificate
- `property EncryptionAlgorithm: string read FEncryptionAlgorithm;` to read the encryption algorithm of the certificate
- `property publicKey: string read FPublicKey;` to read the public key of the certificate
- `property modulus: string read FModulus;` to read the modulus of the certificate
- `property IsSignatureValid: string read FIsSignatureValid;` to read the validity of the certificate
- `property BitSizeEncryptionAlgorithm: Integer read FBitSizeEncryptionAlgorithm write FBitSizeEncryptionAlgorithm;` to read and write the bit size of the encryption algorithm of the certificate
- `property ecCurve: string read FECCurve;` to read the curve type of the certificate
- `property RootCAPath: string read FRootCAPath write FRootCAPath;` to read and write the path to the folder containing the CA certificates
- `property CrtStr: string read FCrtStr write FCrtStr;` to read and write the content in base64 of the certificate (between the ---BEGIN CERTIFICATE--- and ---END CERTIFICATE--- lines)

- `property KeyStr: string read FKeyStr write FKeyStr;` to read and write the content in base64 of the private key (between the ---BEGIN EC PRIVATE KEY--- (or ---BEGIN RSA PRIVATE KEY---) and ---END EC PRIVATE KEY--- (or ---END RSA PRIVATE KEY---) lines)
- `property PSSParam: string read FPSSParam;` to read the RSA PSS parameters of a certificate, i.e. the MGF function and the salt length.

Example of how to generate a self-signed X509 certificate

```
var
  X509Certificate1: TX509Certificate;
begin

  X509Certificate1 := TX509Certificate.Create;
  try
    X509Certificate1.RootCAPath := '.\\RootCA\\';
    X509Certificate1.KeyFilePath := '.\\mykey.key';
    X509Certificate1.CrtFilePath := '.\\mycert.crt';
    X509Certificate1.signatureAlgorithm := TSignAlgo.sa_sha256rsa;
    X509Certificate1.BitSizeEncryptionAlgorithm := 2048;
    X509Certificate1.countryName := 'FR';
    X509Certificate1.stateName := 'Nouvelle-Aquitaine';
    X509Certificate1.localityName := 'Bordeaux';
    X509Certificate1.OrganizationName := 'Cyberens';
    X509Certificate1.commonName := 'Cyberens certificate’;
    X509Certificate1.GenerateSelfSigned;
  Finally
    X509Certificate1.Free;
  end;
end;
```

Example of how to parse an X509 certificate

```
var
  X509Certificate1: TX509Certificate;
  ts: TStringList;
begin
  X509Certificate1 := TX509Certificate.Create;
  ts := TStringList.Create;
  try
    X509Certificate1.RootCAPath := '.\\RootCA\\';
    X509Certificate1.CrtFilePath := '.\\mycert.crt';
    X509Certificate1.Decode;
    ts.Add('Version: ' + X509Certificate1.version);
    ts.Add('Serial number: ' + X509Certificate1.serialNumber);
    ts.Add('Signature algorithm: ' + TabSignAlgo
      [Integer(X509Certificate1.signatureAlgorithm)]);
    ts.Add('not before: ' + X509Certificate1.notBefore);
    ts.Add('not after: ' + X509Certificate1.notAfter);
    ts.Add('Subject country name: ' + X509Certificate1.countryName);
    ts.Add('Subject state or province name: ' + X509Certificate1.stateName);
    ts.Add('Subject locality name: ' + X509Certificate1.localityName);
    ts.Add('Subject organization name: ' + X509Certificate1.OrganizationName);
    ts.Add('Subject organization unit name: ' +
      X509Certificate1.OrganizationUnitName);
    ts.Add('Subject common name: ' + X509Certificate1.commonName);
    ts.Add('Issuer country name: ' + X509Certificate1.IssuercountryName);
```

```
    ts.Add('Issuer state or province name: ' +
      X509Certificate1.IssuerstateName);
    ts.Add('Issuer locality name: ' + X509Certificate1.IssuerlocalityName);
    ts.Add('Issuer organization name: ' +
      X509Certificate1.IssuerOrganizationName);
    ts.Add('Issuer organization unit name: ' +
      X509Certificate1.IssuerOrganizationUnitName);
    ts.Add('Issuer common name: ' + X509Certificate1.IssuerCommonName);
    ts.Add('Alternative name 1: ' + X509Certificate1.AltName1);
    ts.Add('Alternative name 2: ' + X509Certificate1.AltName2);
    ts.Add('Alternative name 3: ' + X509Certificate1.AltName3);
    ts.Add('Alternative name 4: ' + X509Certificate1.AltName4);
    ts.Add('Alternative name 5: ' + X509Certificate1.AltName5);
    ts.Add('Alternative name 6: ' + X509Certificate1.AltName6);
    ts.Add('Asymetric encryption algorithm: ' +
      X509Certificate1.EncryptionAlgorithm + ' ' +
      IntToStr(X509Certificate1.BitSizeEncryptionAlgorithm) + ' bits');
    ts.Add('Modulus: ' + X509Certificate1.Modulus);
    ts.Add('Curve: ' + X509Certificate1.ecCurve);
    ts.Add('Public key: ' + X509Certificate1.publicKey);
    ts.Add(X509Certificate1.IsSignatureValid);
  Finally
    X509Certificate1.Free;
  end;
end;
```

All X509 Certificate generation and parsing functions are located in the X509Obj.pas file.

X509 CSR

CSR means for Certificate Signing Request. It is a message sent from an applicant to a certificate authority in order to apply for a digital identity X.509 certificate. It usually contains the public key for which the certificate should be issued, identifying information (such as a domain name) and integrity protection (e.g., a digital signature). The most common format for CSRs is the PKCS #10 specification.

In this library, you can generate a CSR in PKCS#10 format (only for Desktop platform) and decode them.

The signature algorithms supported by TMS CP are: RSA-SHA256, RSA-SHA384, RSA-SHA512, ECDSA-SHA256 (corresponding to P-256 curve), ECDSA-SHA384 (corresponding to P-384 curve), ECDSA-SHA512 (corresponding to P-521 curve). All RSA algorithms are RSA PKCS#1 v1.5, with a key length of 2048 or 4096 bits.

The X509 CSR class is:

```
TX509CSR = class(TTMSCryptBase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create; reintroduce; overload;
{$IF (defined(MSWINDOWS) or defined(MACOS)) and (not defined(IOS))}
    procedure Generate;
{$IFEND}
    procedure Decode;
  published
    property KeyFilePath: string read FKeyFilePath write setKeyFilePath;
    property CsrFilePath: string read FCsrFilePath write setCsrFilePath;
    property signatureAlgorithm: TSignAlgo read FSignatureAlgorithm
      write setSignatureAlgorithm;
    property hashFunction: TX509HashFunction read FHashfunction write
      SetHashfunction default sha256;
    property countryName: string read FSubjectCountryName write SetCountryName;
    property stateName: string read FSubjectStateName write FSubjectStateName;
    property localityName: string read FSubjectLocalityName
      write FSubjectLocalityName;
    property OrganizationName: string read FSubjectOrganizationName
      write FSubjectOrganizationName;
    property OrganizationUnitName: string read FSubjectOrganizationUnitName
      write FSubjectOrganizationUnitName;
    property commonName: string read FSubjectCommonName write
       FSubjectCommonName;
    property Unicode: TUnicode read FUni write FUni default yesUni;
    property version: string read FVersion;
    property EncryptionAlgorithm: string read FEncryptionAlgorithm;
    property publicKey: string read FPublicKey;
    property modulus: string read FModulus;
    property IsSignatureValid: string read FIsSignatureValid;
    property BitSizeEncryptionAlgorithm: Integer
      read FBitSizeEncryptionAlgorithm write FBitSizeEncryptionAlgorithm;
    property ecCurve: string read FECCurve;
    property CsrStr: string read FCsrStr write FCsrStr;
    property KeyStr: string read FKeyStr write FKeyStr;
  end;
```

The constructors are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `constructor Create; reintroduce; overload;` the default constructor

The public methods are:

- **`procedure`** `Generate;` to generate a CSR (only available on desktop platforms)

- **`procedure`** `Decode;` to decode the fields of a CSR and verify the signature

The properties are:

- `property KeyFilePath: string read FKeyFilePath write setKeyFilePath;` to set and get the path to the private key file
- `property CrtFilePath: string read FCrtFilePath write setCrtFilePath;` to read and write the path to the certificate file
- `property signatureAlgorithm: TSignAlgo read FSignatureAlgorithm write setSignatureAlgorithm;` to read and write the path to the certificate file
- `property hashFunction: TX509HashFunction read FHashfunction write SetHashfunction default sha256;` to read and write the hash function
- `property countryName: string read FSubjectCountryName write SetCountryName;` to read and write the subject country name field
- `property stateName: string read FSubjectStateName write FSubjectStateName;` to read and write the subject state name field
- `property localityName: string read FSubjectLocalityName write FSubjectLocalityName;` to read and write the subject locality name field
- `property OrganizationName: string read FSubjectOrganizationName write FSubjectOrganizationName;` to read and write the subject organization name field
- `property OrganizationUnitName: string read FSubjectOrganizationUnitName write FSubjectOrganizationUnitName;` to read and write the subject organization unit name field
- `property commonName: string read FSubjectCommonName write FSubjectCommonName;` to read and write the subject common name field (mandatory)
- `property Unicode: TUnicode read FUni write FUni default yesUni;` to indicate whether the input buffer or the file name has Unicode characters
- `property version: string read FVersion;` to read the version of the certificate
- `property EncryptionAlgorithm: string read FEncryptionAlgorithm;` to read the encryption algorithm of the certificate
- `property publicKey: string read FPublicKey;` to read the public key of the certificate
- `property modulus: string read FModulus;` to read the modulus of the certificate
- `property IsSignatureValid: string read FIsSignatureValid;` to read the validity of the certificate
- `property BitSizeEncryptionAlgorithm: Integer read FBitSizeEncryptionAlgorithm write FBitSizeEncryptionAlgorithm;` to read and write the bit size of the encryption algorithm of the certificate
- `property ecCurve: string read FECCurve;` to read the curve type of the certificate

- `property CsrStr: string read FCsrStr write FCsrStr;` to read and write the content in base64 of the CSR (between the ---BEGIN CERTIFICATE REQUEST--- and ---END CERTIFICATE REQUEST--- lines)
- `property KeyStr: string read FKeyStr write FKeyStr;` to read and write the content in base64 of the private key (between the ---BEGIN EC PRIVATE KEY--- (or ---BEGIN RSA PRIVATE KEY---) and ---END EC PRIVATE KEY--- (or ---END RSA PRIVATE KEY---) lines)

Example of how to generate a X509 CSR

```
var
  X509CSR1: TX509CSR;
begin

  X509CSR1 := TX509CSR.Create;
  try
    X509CSR1.KeyFilePath := '.\\mykey.key';
    X509CSR1.CsrFilePath := '.\\mycsr.csr';
    X509CSR1.signatureAlgorithm := TSignAlgo.sa_sha256rsa;
    X509CSR1.BitSizeEncryptionAlgorithm := 2048;
    X509CSR1.countryName := 'FR';
    X509CSR1.stateName := 'Nouvelle-Aquitaine';
    X509CSR1.localityName := 'Bordeaux';
    X509CSR1.OrganizationName := 'Cyberens';
    X509CSR1.commonName := 'Cyberens certificate';
    X509CSR1.Generate;
  Finally
    X509CSR1.Free;
  end;
end;
```

Example of how to parse an X509 CSR

```
var
  X509CSR1: TX509CSR;
  ts: TStringList;
begin
  X509CSR1 := TX509CSR.Create;
  ts := TStringList.Create;
  try
    X509CSR1.CsrFilePath := '.\\mycsr.csr';
    X509CSR1.Decode;
    ts.Add('Version: ' + X509CSR1.version);
    ts.Add('Signature algorithm: ' + TabSignAlgo
      [Integer(X509CSR1.signatureAlgorithm)]);
    ts.Add('Subject country name: ' + X509CSR1.countryName);
    ts.Add('Subject state or province name: ' + X509CSR1.stateName);
    ts.Add('Subject locality name: ' + X509CSR1.localityName);
    ts.Add('Subject organization name: ' + X509CSR1.OrganizationName);
    ts.Add('Subject organization unit name: ' +
      X509CSR1.OrganizationUnitName);
    ts.Add('Subject common name: ' + X509CSR1.commonName);
    ts.Add('Asymetric encryption algorithm: ' +
      X509CSR1.EncryptionAlgorithm + ' ' +
      IntToStr(X509CSR1.BitSizeEncryptionAlgorithm) + ' bits');
    ts.Add('Modulus: ' + X509CSR1.Modulus);
    ts.Add('Curve: ' + X509CSR1.ecCurve);
```

```
    ts.Add('Public key: ' + X509CSR1.publicKey);
    ts.Add(X509CSR1.IsSignatureValid);
  Finally
    X509CSR1.Free;
  end;
end;
```

All X509 CSR generation and parsing functions are located in the X509Obj.pas file.

PKCS11

PKCS#11 is the programming interface to create and manipulate cryptographic tokens.

The PKCS #11 standard defines a platform-independent API to cryptographic tokens, such as hardware security modules (HSM) and smart cards, and names the API itself "Cryptoki" (from "cryptographic token interface" and pronounced as "crypto-key" - but "PKCS #11" is often used to refer to the API as well as the standard that defines it).

The API defines most commonly used cryptographic object types (RSA keys, X.509 Certificates, AES keys, etc.) and all the functions needed to use, create/generate, modify and delete those objects.

Each token has a different DLL filename to access to PKCS11 library functions. You need to know the filename of your driver to use our component. There is a list of known driver filenames here:

http://wiki.ncryptoki.com/Known-PKCS-11-modules.ashx

Obviously, you need to know the PIN code of your token if you would like to use secret or private keys.

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of "slots". Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is "present in the slot" (typically) when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots and applications can connect to all those tokens.

In our library, we suppose that one slot = one token. To use the component, you need to precise which slot index you want to use into the list of available slots. The “first” token into this slot is automatically selected. Tell us if you have more than one token into a slot.

In TMS Cryptography Pack, there are 3 files for PKCS11:

- PKCS11Values.pas that contains all Cryptoki constants and types
- PKCS11Library.pas that contains all Cryptoki basic functions, the documentation of these functions is available here: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html
- PKCS11Obj.pas that contains a component class to use easily the Cryptoki functions. This class is TPKCS11, described below. You can ONLY use it on Windows platforms.

```
type
  TPKCS11param = packed record
    isToken: boolean;
    SlotIndex: Integer;
    CertIndex: Integer;
    DLLPath: string;
```

```
    Pin: string;
  end;

TPKCS11 = class(TTMSCryptBase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create; reintroduce; overload;
    constructor Create(DLLPath: string); reintroduce; overload;

    function ListSlots: TStringList;
    function ListTokens: TStringList;
    function ListObjects: TStringList;
    function ListCertificates: TStringList;
    function ListPrivateKeys: TStringList;
    function ListPublicKeys: TStringList;
    function ListSecretKeys: TStringList;
    function ListMechanisms: TStringList;

    procedure OpenSession;
    procedure CloseSession;
    procedure Login(PIN: string);
    procedure Logout;
    procedure SetPIN(oldPin: string; newPIN: string);
    procedure InitPIN(PIN: string);

    function CertificatesIndex: TIndexArray;
    function SecretKeysIndex: TIndexArray;
    function PrivateKeysIndex: TIndexArray;
    function PublicKeysIndex: TIndexArray;

    function PrivateKeyIndexFromID(id: string): Integer;
    function PublicKeyIndexFromID(id: string): Integer;
    function CertificateIndexFromID(id: string): Integer;
    function GetObjectID(index: Integer): string;
    function ExtractCertificate(index: Integer): string;

    function SignWithPrivateKey(s: string): string;
    function SignWithSecretKey(s: string; algorithm: TMACAlgorithm): string;
    function VerifyWithPublicKey(s, signature: string): Integer;
    function VerifyWithSecretKey(s, signature: string;
      algorithm: TMACAlgorithm): Integer;

    function EncryptWithPublicKey(s: string; algo: TAsymEncAlgo): string;
    function AESEncryptWithSecretKey(s: string; mode: TAESMode;
      IV: string): string;
    function DecryptWithPrivateKey(s: string; algo: TAsymEncAlgo): string;
    function AESDecryptWithSecretKey(s: string; mode: TAESMode;
      IV: string): string;

    function ShowKey(index: Integer): TStringList;
    function ShowCert(index: Integer): TStringList;

    procedure GenerateAESKey(keyLength: Integer);
    procedure GenerateGenericSecretKey(keyLength: Integer);

    function IsCertificate(index: Integer): boolean;
    function IsPublicKey(index: Integer): boolean;
    function IsPrivateKey(index: Integer): boolean;
```

```
    function IsSecretKey(index: Integer): boolean;

  published
    property currentSlotIndex: Integer read FcurrentSlotIndex
      write setCurrentSlot default -1;
    property currentObjectIndex: Integer read FCurrentObjectIndex
      write setCurrentObject default -1;
    property outputFormat: TConvertType read FoutputFormat write FoutputFormat
      default base64;
    property DLLpath: string read FDLLPath write setDLLPath;
  end;
```

The constructors are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `constructor Create; reintroduce; overload;` the default constructor
- `constructor Create(DLLPath: string); reintroduce; overload;` the constructor to set the DLL file path, required to access to the token library

The public methods are:

- **function** `ListSlots: TStringList;` to list the slots, i.e. the cryptographic devices that are active
- **function** `ListTokens: TStringList;` to list the tokens that are present on the current slot. We suppose that one slot = one token.
- **function** `ListObjects: TStringList;` to list all objects on the current slot
- **function** `ListCertificates: TStringList;` to get the details of the certificates present on the current slot
- **function** `ListPrivateKeys: TStringList;` to get the details of the private keys present on the current slot
- **function** `ListPublicKeys: TStringList;` to get the details of the public keys present on the current slot
- **function** `ListSecretKeys: TStringList;` to get the details of the secret keys present on the current slot
- **function** `ListMechanisms: TStringList;` to get the list of the mechanisms of the token, i.e. the available algorithms.
- **procedure** `OpenSession;` to open a session
- **procedure** `CloseSession;` to close a session
- **procedure** `Login(PIN: string);` to log in
- **procedure** `Logout;` to log out
- **procedure** `SetPIN(oldPin: string; newPIN: string);` to change the PIN
- **procedure** `InitPIN(PIN: string);` to set the PIN if the token does not have one
- **function** `CertificatesIndex: TIndexArray;` to get the index of certificates in list objects
- **function** `SecretKeysIndex: TIndexArray;` to get the index of secret keys in list objects
- **function** `PrivateKeysIndex: TIndexArray;` to get the index of private keys in list objects
- **function** `PublicKeysIndex: TIndexArray;` to get the index of public keys in list objects

- `function PrivateKeyIndexFromID(id: string): Integer;` to get the index of a private key from its ID
- `function PublicKeyIndexFromID(id: string): Integer;` to get the index of a public key from its ID
- `function CertificateIndexFromID(id: string): Integer;` to get the index of a certificate from its ID
- `function GetObjectID(index: Integer): string;` to get the ID of an object
- `function ExtractCertificate(index: Integer): string;` to extract the certificate in base64 format
- `function SignWithPrivateKey(s: string): string;` to sign the string s with the current private key (see property currentObjectIndex to set the private key)
- `function SignWithSecretKey(s: string; algorithm: TMACAlgorithm): string;` to sign the string s with the current secret key (see property currentObjectIndex to set the private key) and using a MAC algorithm
- `function VerifyWithPublicKey(s, signature: string): Integer;` to verify the signature of the string s with the current public key (see property currentObjectIndex to set the public key)
- `function VerifyWithSecretKey(s, signature: string; algorithm: TMACAlgorithm): Integer;` to verify the signature of the string s with the current secret key (see property currentObjectIndex to set the secret key) and using a MAC algorithm
- `function EncryptWithPublicKey(s: string; algo: TAsymEncAlgo): string;` to encrypt a string s with the current public key (see property currentObjectIndex to set the public key) and using an asymmetric encryption algorithm
- `function AESEncryptWithSecretKey(s: string; mode: TAESMode; IV: string): string;` to encrypt a string s with the current secret key (see property currentObjectIndex to set the secret key) and using AES with a mode and an IV (if not ECB mode).
- `function DecryptWithPrivateKey(s: string; algo: TAsymEncAlgo): string;` to decrypt a string s with the current private key (see property currentObjectIndex to set the private key) and using an asymmetric encryption algorithm
- `function AESDecryptWithSecretKey(s: string; mode: TAESMode; IV: string): string;` to decrypt a string s with the current secret key (see property currentObjectIndex to set the secret key) and using AES with a mode and an IV (if not ECB mode).
- `function ShowKey(index: Integer): TStringList;` to show the details of a key (public, secret or public)
- `function ShowCert(index: Integer): TStringList;` to show the details of a certificate
- `procedure GenerateAESKey(keyLength: Integer);` to generate a non-persistent AES key on the token.
- `procedure GenerateGenericSecretKey(keyLength: Integer);` to generate a non-persistent Generic secret key on the token.
- `function IsCertificate(index: Integer): boolean;` to know whether the object number index is a certificate
- `function IsPublicKey(index: Integer): boolean;` to know whether the object number index is a public key
- `function IsPrivateKey(index: Integer): boolean;` to know whether the object number index is a private key

- `function IsSecretKey(index: Integer): boolean;` to know whether the object number index is a secret key

The properties are:

- `property currentSlotIndex: Integer read FcurrentSlotIndex write setCurrentSlot default -1;` to get and set the current slot index
- `property currentObjectIndex: Integer read FCurrentObjectIndex write setCurrentObject default -1;` to get and set the current object (private key, public key, secret key or certificate)
- `property outputFormat: TConvertType read FoutputFormat write FoutputFormat default base64;` to get and set the outputFormat (also the format of the signature and the decrypt input)
- `property DLLpath: string read FDLLPath write setDLLPath;` to get and set the path fo the DLL filename

Example

```
var
  p11: TPKCS11;
  ts, ts2: TStringList;
  I, j: integer;
begin

  p11 := TPKCS11.Create('aetpkss1.dll');
  try
    p11.currentSlotIndex := 0;
    p11.Login('123456');
    ts := TStringList.Create;
    try
      ts := p11.ListObjects;
      for i := 0 to ts.Count - 1 do
        Memo1.Lines.Add(ts.Strings[i]);
      for i := 0 to ts.Count - 1 do
      begin
        if p11.IsPrivateKey(i) or p11.IsSecretKey(i) or p11.IsPublicKey(i) then
        begin
          ts2 := p11.ShowKey(i);
          try
            for j := 0 to ts2.Count - 1 do
              Memo1.Lines.Add(ts2.Strings[j]);
          Finally
            ts2.Free;
          end;
        end;
        if p11.IsCertificate(i) then
        begin
          ts2 := p11.ShowCert(i);
          try
            for j := 0 to ts2.Count - 1 do
              Memo1.Lines.Add(ts.Strings[j]);
          Finally
            ts2.Free;
          end;
        end;
```

```
      end;
    Finally
      ts.Free;
    end;
  Finally
    p11.Free;
  end;
end;
```

The PKCS11Param record is used into the XAdES, CAdES and PAdES class to set the required property to sign documents with cryptographic token.

XAdES

XAdES (short for "**XML Advanced Electronic Signatures**") is a set of extensions to XML-DSig recommendation making it suitable for advanced electronic signatures. W3C and ETSI maintain and update XAdES together.

While XML-DSig is a general framework for digitally signing documents, XAdES specifies precise profiles of XML-DSig making it compliant with the European eIDAS regulation (*Regulation on electronic identification and trust services for electronic transactions in the internal market*). EIDAS is legally binding in all EU member states since July 2014. An electronic signature that has been created in compliance with eIDAS has the same legal value as a handwritten signature.

There are 6 profiles for XAdES : XAdES-BES, XAdES-T, XAdES-C, XAdES-X, XAdES-X-L, XAdES-A. TMS Cryptography Pack supports only the XAdES-BES profile.

There are 3 signature formats:
- detached: the signature is detached from the document to sign.
- enveloping: the document to sign is added to the signature.
- enveloped: the signature is added to the XML document to sign.

The XAdES class is:

```
TSignatureMethod = (rsasha1, rsasha256, rsasha384, rsasha512, ecdsasha256,
    ecdsasha384, ecdsasha512);
TDigestMethod = (sha256, sha384, sha512);
TPackaging = (detached, enveloping, enveloped);

TXAdES = class(TTMSCryptBase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create; reintroduce; overload;
    constructor Create(Cert: TX509Certificate); reintroduce;
        overload;
    constructor Create(KeyFilePath: string; CertPath: string); reintroduce;
        overload;
    destructor Destroy; override;
    procedure GenerateSignature(FilePath: string; OutputFilePath: string);
    procedure ChangeSignatureTagLocation(SignatureFilePath: string;
      parentTag: string; AdditionalParentTags: TStringList;
      OutputFilePath: string);
    function VerifySignature(FilePath: string): integer;
    function VerifyError(err: Integer): string;
{$IF defined(MSWINDOWS)}
    procedure LoadCertAndKeyFromPKCS12(FilePath: string; Password: string;
        PathToOpenSSL: string);
{$IFEND}
    function GetFileMIMEType(const AfileName: String): String;
{$IF defined(MSWINDOWS)}
    property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;
{$IFEND}
  published
    property SignatureMethod: TSignatureMethod read FSignatureMethod;
    property KeyFilePath: string read FKeyFilePath write FKeyFilePath;
    property Packaging: TPackaging read FPackaging
```

```
      write FPackaging default enveloping;
    property CertFilePath: string read FCertFilePath write setCertFilePath;
    property ErrorDetails: string read FErrorDetails;
    property PathToOriginalFile: string read FPathToOriginalFile write
       FPathToOriginalFile;
    property Progress: integer read FProgress write SetProgress default 0;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  end;
```

The constructors are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `constructor Create; reintroduce; overload;` the default constructor
- `constructor Create(Cert: TX509Certificate); reintroduce; overload;` the constructor to set the X509 certificate

- `constructor Create(KeyFilePath: string; CertPath: string); reintroduce; overload;` the constructor to set the path to the private key file and the path to the certificate file

The public methods are:

- **procedure** `GenerateSignature(FilePath: string; OutputFilePath: string);` to generate an XAdES signature of FilePath in the file OutputFilePath

- **procedure** `ChangeSignatureTagLocation(SignatureFilePath: string; parentTag: string; AdditionalParentTags: TStringList; OutputFilePath: string);` to change the location of the signature, i.e. to put the signature into parentTag with AdditionalParentTags as parents.
- **function** `VerifySignature(FilePath: string): integer;` to verify the signature

- **function** `VerifyError(err: Integer): string;` to display the error associated to the error code err
- **procedure** `LoadCertAndKeyFromPKCS12(FilePath: string; Password: string; PathToOpenSSL: string);` to import a pfx encrypted file. We need the password to decrypt it and the folder path to openssl.exe file. This function is available only on Windows.
- **function** `GetFileMIMEType(const AfileName: String): String;` to know the MIME type of a file

The properties are:

- `property SignatureMethod: TSignatureMethod read FSignatureMethod;` to read the signature method
- `property KeyFilePath: string read FKeyFilePath write FKeyFilePath;` to read and write the path to the private key file
- `property Packaging: TPackaging read FPackaging write FPackaging default enveloping;` to read and write the packaging detached, enveloping or enveloped
- `property CertFilePath: string read FCertFilePath write SetCertPath;` to read and write the path to the certificate file

- `property ErrorDetails: string read FErrorDetails;` to read the error details of verifying signature
- `property PathToOriginalFile: string read FPathToOriginalFile write FPathToOriginalFile;` to read and write the path to the folder containing the original file
- `property Progress: integer read FProgress write SetProgress default 0;` to set and get the progress of the generation or verification process
- `property OnChange: TNotifyEvent read FOnChange write FOnChange;` to indicate that the progress changes
- `property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;` to get and set the required parameters to use the XAdES object with a cryptographic token

Example of how to generate a XAdES signature

```
var
  XAdES: TXAdES;
begin

  XAdES := TXAdES.Create;
  try
    XAdES.KeyFilePath:= '.\\mykey.key';
    XAdES.CertFilePath:= '.\\mycert.crt';
    XAdES.Packaging := enveloping;
    XAdES.PathToOriginalFile := ExtractFilePath(XAdES.KeyFilePath);
    XAdES.GenerateSignature('.\\test.txt', '.\\signature.xml');
  Finally
    XAdES.Free;
  end;
end;
```

Example of how to verify an XAdES signature

```
var
  XAdES: TXAdES;
  Err : Integer;
  S, t: string;
begin
  XAdES := TXAdES.Create;
  try
    XAdES.PathToOriginalFile := '.';
    err := XAdES.VerifySignature('.\\signature.xml');
    s:= XAdES.VerifyError(err);
    if err < 0 then
        t := XAdES.ErrorDetails;
  Finally
    XAdES.Free;
  end;
end;
```

Example of how to sign an XML document with a cryptographic token

```
var
```

```
  XAdES: TXAdES;
  Err : Integer;
  S, t: string;
  PKCS11: TPKCS11;
begin
  XAdES := TXAdES.Create;
  try
    XAdES.Packaging := enveloped;
    XAdES.PKCS11Param.isToken := true;
    XAdES.PKCS11Param.SlotIndex := 0;
    PKCS11 := TPKCS11.Create('aetpkss1.dll');
    try
      XAdES.PKCS11Param.CertIndex := PKCS11.CertificatesIndex[0];
    finally
      PKCS11.Free;
    end;
    XAdES.PKCS11Param.DLLPath := 'aetpkss1.dll';
    XAdES.PKCS11Param.Pin := '123456';
    XAdES.GenerateSignature('.\\test.xml', '.\\test_with_signature.xml');
  Finally
    XAdES.Free;
  end;
end;
```

All XAdES generation and verifying functions are located in the XAdESObj.pas file.

CAdES

CAdES (short for "**CMS Advanced Electronic Signatures**") is a set of extensions to Cryptographic Message Syntax (CMS) signed data making it suitable for advanced electronic signatures. ETSI maintains and updates CAdES.

CMS is a general framework for Electronic Signatures for various kinds of transactions like purchase requisition, contracts or invoices. CAdES specifies precise profiles of CMS signed data making it compliant with the European eIDAS regulation (Regulation on electronic identification and trust services for electronic transactions in the internal market).

There are 4 profiles for CAdES: CAdES-B, CAdES-T, CAdES-LT, CAdES-LTA. TMS Cryptography Pack supports only the CAdES-B profile.

There are 2 signature formats:
- detached: the signature is detached from the document to sign.
- enveloping: the document to sign is added to the signature.

The CAdES class is:

```
TSignatureMethod = (rsasha1, rsasha256, rsasha384, rsasha512, ecdsasha256,
    ecdsasha384, ecdsasha512);
TDigestMethod = (sha256, sha384, sha512);
TPackaging = (detached, enveloping, enveloped);

TCAdES = class(TTMSCryptBase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(KeyFilePath: string; CertPath: string);
      reintroduce; overload;
    constructor Create(KeyFilePath: string; CertPath: string; fp: boolean);
      reintroduce; overload;
    constructor Create; reintroduce; overload;
    constructor Create(fp: boolean); reintroduce; overload;
    constructor Create(Cert: TX509Certificate; fp: boolean); reintroduce;
        overload;
    function GenerateSignature(FilePath: string; OutputFilePath: string;
      AdditionalData: string): string;
    function VerifySignature(FilePath: string; OriginalFile: string): integer;
    function VerifyError(err: integer): string;
{$IF defined(MSWINDOWS)}
    procedure LoadCertAndKeyFromPKCS12(FilePath: string; Password: string;
        PathToOpenSSL: string);
{$IFEND}
    function GetFileMIMEType(const AfileName: String): String;
{$IF defined(MSWINDOWS)}
    property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;
{$IFEND}
  published
    property SignatureMethod: TSignatureMethod read FSignatureMethod;
    property KeyFilePath: string read FKeyFilePath write FKeyFilePath;
    property Packaging: TPackaging read FPackaging
      write FPackaging default enveloping;
    property CertFilePath: string read FCertPath write setCertPath;
```

```
    property ErrorDetails: string read FErrorDetails;
    property PathToOriginalFile: string read FPathToOriginalFile write
        FPathToOriginalFile;
    property Progress: integer read FProgress write SetProgress default 0;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
    property ForPAdES: boolean read FForPAdES write FForPAdES;
  end;
```

The constructors are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `constructor Create; reintroduce; overload;` the default constructor
- `constructor Create(KeyFilePath: string; CertPath: string); reintroduce; overload;` the constructor to set the path to the private key file and the path to the certificate file

- `constructor Create(KeyFilePath: string; CertPath: string; fp: boolean); reintroduce; overload;` the constructor to set the path to the private key file, the path to the certificate file and the ForPAdES property

- `constructor Create(fp: boolean); reintroduce; overload;` the constructor to set the ForPAdES property

- `constructor Create(cert: TX509Certificate; fp: boolean); reintroduce; overload;` the constructor to set the X509 certificate and the ForPAdES property

The public methods are:

- **function** GenerateSignature`(FilePath: string; OutputFilePath: string; AdditionalData: string): string`; to generate a CAdES signature of FilePath concatenated to AdditionalData in the file OutputFilePath
- **function** `VerifySignature(FilePath: string; OriginalFile: string): integer;` to verify the signature

- **function** `VerifyError(err: Integer): string;` to display the error associated to the error code err
- **procedure** `LoadCertAndKeyFromPKCS12(FilePath: string; Password: string; PathToOpenSSL: string);` to import a pfx encrypted file. We need the password to decrypt it and the folder path to the openssl.exe file. This function is available only on Windows.
- **function** `GetFileMIMEType(const AfileName: String): String;` to know the MIME type of a file

The properties are:

- `property SignatureMethod: TSignatureMethod read FSignatureMethod;` to read the signature method
- `property KeyFilePath: string read FKeyFilePath write FKeyFilePath;` to read and write the path to the private key file

- `property Packaging: TPackaging read FPackaging write FPackaging default` enveloping; to read and write the packaging detached, enveloping or enveloped
- `property CertFilePath: string read FCertFilePath write SetCertFilePath;` to read and write the path to the certificate file
- `property ErrorDetails: string read FErrorDetails;` to read the error details of verifying signature
- `property PathToOriginalFile: string read FPathToOriginalFile write FPathToOriginalFile;` to read and write the path to the folder containing the original file
- `property Progress: integer read FProgress write SetProgress default 0;` to set and get the progress of the generation or verification process
- `property OnChange: TNotifyEvent read FOnChange write FOnChange;` to indicate that the progress changes
- `property ForPAdES: boolean read FForPAdES write FForPAdES;` to indicate whether the CAdES signature is used for PAdES
- `property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;` to get and set the required parameters to use the CAdES object with a cryptographic token

Example of how to generate a CAdES signature

```
var
  CAdES: TCAdES;
begin

  CAdES := TCAdES.Create;
  try
    CAdES.KeyFilePath:= '.\\mykey.key';
    CAdES.CertFilePath:= '.\\mycert.crt';
    CAdES.Packaging := enveloping;
    CAdES.GenerateSignature('.\\test.txt', '.\\signature.p7m');
  Finally
    CAdES.Free;
  end;
end;
```

Example of how to verify a CAdES signature

```
var
  CAdES: TCAdES;
  Err : Integer;
  S, t: string;
begin
  CAdES := TCAdES.Create;
  try
    err := CAdES.VerifySignature('.\\signature.p7m', '.\\test.txt');
    s:= CAdES.VerifyError(err);
    if err < 0 then
        t := XAdES.ErrorDetails;
  Finally
    CAdES.Free;
  end;
end;
```

Example of how to sign a document with a cryptographic token

```
var
  CAdES: TCAdES;
  Err : Integer;
  S, t: string;
  PKCS11: TPKCS11;
begin
  CAdES := TCAdES.Create;
 try
    CAdES.Packaging := enveloping;
    CAdES.PKCS11Param.isToken := true;
    CAdES.PKCS11Param.SlotIndex := 0;
    PKCS11 := TPKCS11.Create('aetpkss1.dll');
    try
      CAdES.PKCS11Param.CertIndex := PKCS11.CertificatesIndex[0];
    finally
      PKCS11.Free;
    end;
    CAdES.PKCS11Param.DLLPath := 'aetpkss1.dll';
    CAdES.PKCS11Param.Pin := '123456';
    CAdES.GenerateSignature('.\\test.txt', '.\\signature.p7m');
  Finally
    CAdES.Free;
  end;
end;
```

All CAdES generation and verifying functions are located in the CAdESObj.pas file.

PAdES

PAdES (*PDF Advanced Electronic Signatures*) is a set of restrictions and extensions to PDF and ISO 32000-1 making it suitable for Advanced Electronic Signature. This is published by ETSI as TS 102 778.

While PDF and ISO 32000-1 provide a framework for digitally signing their documents, PAdES specifies precise profiles making it compliant with the European eIDAS regulation (Regulation on electronic identification and trust services for electronic transactions in the internal market).

To sign a PDF by several signers, you need to sign original PDF by signer 1. Then you need to sign the signed PDF by signer 2, etc.

The PAdES class is:

```
TSignatureMethod = (rsasha1, rsasha256, rsasha384, rsasha512, ecdsasha256,
    ecdsasha384, ecdsasha512);
TDigestMethod = (sha256, sha384, sha512);

TPAdES = class(TTMSCryptBase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(KeyFilePath: string; CertPath: string);
      reintroduce; overload;
    constructor Create; reintroduce; overload;
    destructor Destroy; override;
    procedure GenerateSignature(FilePath: string; OutputFilePath: string);
       overload;
    procedure GenerateSignature(InStream: TStream; OutStream: TStream);
       overload;
    function VerifySignature(FilePath: string; OriginalFile: string): integer;
    function VerifyError(err: integer): string;
{$IF defined(MSWINDOWS)}
    procedure LoadCertAndKeyFromPKCS12(FilePath: string; Password: string;
       PathToOpenSSL: string);
{$IFEND}
    function GetFileMIMEType(const AfileName: String): String;
{$IF defined(MSWINDOWS)}
    property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;
{$IFEND}
  published
    property SignatureMethod: TSignatureMethod read FSignatureMethod;
    property KeyFilePath: string read FKeyFilePath write FKeyFilePath;
    property CertFilePath: string read FCertPath write setCertPath;
    property ErrorDetails: string read FErrorDetails;
    property PathToOriginalFile: string read FPathToOriginalFile write
       FPathToOriginalFile;
    property Progress: integer read FProgress write SetProgress default 0;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  end;
```

The constructors and destructor are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class

- `constructor Create; reintroduce; overload;` the default constructor
- `constructor Create(KeyFilePath: string; CertPath: string); reintroduce; overload;` the constructor to set the path to the private key file and the path to the certificate file

- `destructor Destroy; override;` the default destructor

The public methods are:

- **procedure** GenerateSignature`(FilePath: string; OutputFilePath: string);` to generate a PAdES signature of FilePath in the file OutputFilePath
- **procedure** GenerateSignature`(InStream: TStream; OutStream: TStream);` to generate a PAdES signature of InStream in the stream OutStream
- **function** `VerifySignature(FilePath: string; OriginalFile: string): integer;` to verify the signature

- **function** `VerifyError(err: Integer): string;` to display the error associated to the error code err
- **procedure** `LoadCertAndKeyFromPKCS12(FilePath: string; Password: string; PathToOpenSSL: string);` to import a pfx encrypted file. We need the password to decrypt it and the folder path to the openssl.exe file. This function is available only on Windows.
- **function** `GetFileMIMEType(const AfileName: String): String;` to know the MIME type of a file

The properties are:

- `property SignatureMethod: TSignatureMethod read FSignatureMethod;` to read the signature method
- `property KeyFilePath: string read FKeyFilePath write FKeyFilePath;` to read and write the path to the private key file
- `property Packaging: TPackaging read FPackaging write FPackaging default` enveloping`;` to read and write the packaging detached, enveloping or enveloped
- `property CertFilePath: string read FCertFilePath write SetCertPath;` to read and write the path to the certificate file
- `property ErrorDetails: string read FErrorDetails;` to read the error details of verifying signature
- `property PathToOriginalFile: string read FPathToOriginalFile write FPathToOriginalFile;` to read and write the path to the folder containing the original file
- `property Progress: integer read FProgress write SetProgress default 0;` to set and get the progress of the generation or verification process
- `property OnChange: TNotifyEvent read FOnChange write FOnChange;` to indicate that the progress changes
- `property PKCS11Param: PPKCS11Param read GetPKCS11Param write SetPKCS11Param;` to get and set the required parameters to use the PAdES object with a cryptographic token

Example of how to generate a PAdES signature

```
var
  PAdES: TPAdES;
begin

  PAdES := TPAdES.Create;
  try
    PAdES.KeyFilePath:= '.\\mykey.key';
    PAdES.CertFilePath:= '.\\mycert.crt';
    PAdES.GenerateSignature('.\\test.pdf', '.\\signature.pdf');
  Finally
    PAdES.Free;
  end;
end;
```

Example of how to verify a PAdES signature

```
var
  PAdES: TPAdES;
  Err : Integer;
  S, t: string;
begin
  PAdES := TPAdES.Create;
  Try
    err := PAdES.VerifySignature('.\\signature.pdf');
    s:= PAdES.VerifyError(err);
    if err < 0 then
        t := PAdES.ErrorDetails;
  Finally
    PAdES.Free;
  end;
end;
```

Example of how to sign a PDF document with a cryptographic token

```
var
  PAdES: TPAdES;
  Err : Integer;
  S, t: string;
  PKCS11: TPKCS11;
begin
  PAdES := TPAdES.Create;
 try
    PAdES.Packaging := enveloped;
    PAdES.PKCS11Param.isToken := true;
    PAdES.PKCS11Param.SlotIndex := 0;
    PKCS11 := TPKCS11.Create('aetpkss1.dll');
    try
      PAdES.PKCS11Param.CertIndex := PKCS11.CertificatesIndex[0];
    finally
      PKCS11.Free;
    end;
    PAdES.PKCS11Param.DLLPath := 'aetpkss1.dll';
    PAdES.PKCS11Param.Pin := '123456';
    PAdES.GenerateSignature('.\\test.pdf', '.\\test_with_signature.pdf');
```

```
  Finally
    PAdES.Free;
  end;
end;
```

All PAdES generation and verifying functions are located in the PAdESObj.pas file.

Random generators

To generate random integers or random buffers, you can use the following functions (in MiscObj.pas).
On Windows or OSX:

- `function RandomBuffer(len: Integer; MyBuffer: PAnsiChar): Integer;`
- `function RandomUBuffer(len: Integer; MyBuffer: PAnsiChar): Integer;`
- `function RandomInt: Integer;`
- `function RandomUInt: Integer;`

On iOS or Android:

- `function RandomBuffer(len: Integer; MyBuffer: PByte): Integer;`
- `function RandomUBuffer(len: Integer; MyBuffer: PByte): Integer;`
- `function RandomInt: Integer;`
- `function RandomUInt: Integer;`

RandomBuffer and RandomUBuffer fill the buffer MyBuffer with len random characters (and return an error if it fails). On Windows, the functions use the same algorithm, but in the other targets, they use /dev/random for RandomBuffer and /dev/urandom for RandomUBuffer. So, if you want to generate some cryptographic keys, preferably use RandomBuffer, and use RandomUBuffer for salt, IV, or other data which are not keys. We recommend the same for RandomInt and RandomUInt.

Moreover, you can generate a random string by using the RandomString method of TConvert class.

All RANDOM functions/procedures are located in the MiscObj file.

Encrypt an ini file

Included in the TMS Cryptography Pack is also the non-visual class TEncryptedIniFile that offers the capability to store application settings in an encrypted INI file. TEncryptedIniFile descends from TMemInifile, so it inherits all methods to read and write various types (string, number, Boolean, …) to an INI file. The encryption and decryption is done in memory, so at no time, the file system 'sees' an unencrypted file. TEncryptedIniFile uses internally AES 256bit encryption. Further, the only difference with a regular TINIFile class is the added encryption key parameter in the constructor of TEncryptedIniFile.

The class definition looks like:

```
TEncryptedIniFile = class(TMemInifile)
  private
    FFileName: string;
    FEncoding: TEncoding;
    FKey: string;
    FOnDecryptError: TNotifyEvent;
    FUni: TUnicode;
    FOutputFormat: TConvertType;
    procedure LoadValues;
  public
    constructor Create(const FileName: string; const Key: string); overload;
    constructor Create(const FileName: string; const Encoding: TEncoding;
      CaseSensitive: Boolean); overload; override;
    constructor Create(const FileName: string; const Key: string; const Uni:
      TUnicode; const OutputFormat: TConvertType); overload; override;
    procedure UpdateFile; override;
  published
    property OnDecryptError: TNotifyEvent read FOnDecryptError
      write FOnDecryptError;
  end;
```

A sample to use this class to read data back from such encrypted file is here:

```
const
  aeskey = 'anijd54dee1c3e87e1de1d6e4d4e1de3';
var
  mi: TEncryptedIniFile;
begin
  try
    mi := TEncryptedIniFile.Create('.settings.cfg', aeskey);
    try
      FTPUserNameEdit.Text := mi.ReadString('FTP','USER','');
      FTPPasswordNameEdit.Text := mi.ReadString('FTP','PWD','');
      FTPPortSpin.Value := mi.ReadInteger('FTP','PORT',21);
      mi.WriteDateTime('SETTINGS','LASTUSE',Now);
      mi.UpdateFile;
   finally
      mi.Free;
   end;
  except
    ShowMessage('Error in encrypted file. Someone tampered with the file?');
  end;
```

```
end;
```

To ensure backward compatibility with TMS CP 2.5.1 and older, we added a third constructor:

```
constructor Create(const FileName: string; const Key: string; const Uni:
TUnicode; const OutputFormat: TConvertType); overload; override;
```

An ini file encrypted with TMS CP 2.5.1 can be decrypted by using the following code:

```
const
  aeskey = 'anijd54dee1c3e87e1de1d6e4d4e1de3';
var
  mi: TEncryptedIniFile;
begin
  try
    mi := TEncryptedIniFile.Create('.settings.cfg', aeskey, noUni, base64);
  except
    MessageDlg('Error: ' + mi, mtError, [mbOK], 0);
  end;
  FreeAndNil(mi);
end;
```

The TEncryptedIniFile class is located in the TMSEncryptedIniFile file.

Generate a self-decrypted file

Included in the TMS Cryptography Pack is also the component TLockFile that offers the capability to generate a self-decrypted executable, i.e. a file able to decrypt itself. The user chooses a file and a password. The execute method encrypts this file with the password and add it as a resource of an executable (loaded from the resource of the current program) able to decrypt the file. The process uses AES-256, Argon2 and SHA-256. This component is available only on Windows platform.

The component uses a resource named unlockfileres.RES where the unlocking executable is loaded.

You can find a demo of TLockFile in the TMS Software website :
https://www.tmssoftware.com/site/freetools.asp#lockfile

The component definition looks like:

```
TLockFile = class(TTMSCryptbase)
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create; reintroduce; override;
    constructor Create(UnlockFilePath: string); reintroduce; override;
    destructor Destroy; override;
    procedure Execute(encFile, password, outputFile: string);
  published
    property Progress: Integer read FProgress write setProgress default 0;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
    property UnlockFileExePath: string read FUnlockFileExePath
      write FUnlockFileExePath;
  end;
```

The constructors and destructor are:

- `constructor Create(AOwner: TComponent); overload; override;` the default constructor from the TComponent class
- `constructor Create; reintroduce; overload;` the default constructor
- `constructor Create(UnlockFilePath: string); reintroduce; overload;` the constructor to set the path to the loaded resource

- `destructor Destroy; override;` the default destructor

The public methods are:

- **procedure** `Execute(encFile, password, outputFile: string);` to encrypt the file encFile with the password and save the generated executable in outputFile.

The properties are:

- `property Progress: integer read FProgress write SetProgress default 0;` to set and get the progress of the locking process
- `property OnChange: TNotifyEvent read FOnChange write FOnChange;` to indicate that the progress changes

- `property UnlockFileExePath: string read FUnlockFileExePath write FUnlockFileExePath;` to read and write the path to the folder containing the loaded resource

Example of how to use TLockFile

```
var
  TMSLockFile: TLockFile;
begin
  TMSLockFile := TLockFile.Create;
  Try
    TMSLockFile.UnlockFileExePath := TPath.GetHomePath;
    TMSLockFile.Execute('.\\file.txt', 'password123!', '.\\file_encrypted.exe');
  Finally
    TMSLockFile.Free;
  end;
end;
```

Troubleshooting

There are several potential issues when running the various demos included in TMS Cryptography Pack.

RandomDLL.DLL

It is necessary to copy this DLL in the appropriate folder to run the Windows 64 demo and to use the library in Windows 64 applications for RAD Studio version under 10.2.1.
Copy RandomDLL.dll from the Win64 directory of TMS Cryptography Pack:
- to C:\Windows\SysWOW64 if you are running 32 bit Windows
- or to C:\Windows\System32 if you are running 64 bit Windows

For versions after 10.2.1, you can bypass the use of RandomDLL.dll by uncommenting the line // {$DEFINE IDEVERSION1021} in tmscrypto.inc file.

libTMSCPLib.a

Some error messages contain "… libTMSCPLib.a not found". In this case, the search path for the libraries needs to be updated. Go to Project->Options, then Search Path and click on "…" to update your list with the directory location of libTMSCPLib.a (for instance "FULL TMS INSTALLATION PATH\iOSDevice64").

C++ demo

To use the C++ demo, you need to add the .a file to the project for Android or iOS target.

iOS Simulator

The TMS Cryptography Pack does not support the iOS Simulator because we generate .a files from C code and we cannot generate .a file for iOS Simulator target with RAD Studio.

Import a public/private key from an OpenSSL file

To do use the TRSAEncSign methods to import a public/private key from an OpenSSL file, you must have OpenSSL installed on Windows or OSX. For Android/iOS, OpenSSL is included in the libcrypto.a/libcrypto.so.1.0.0 and libssl.a/libssl.so.1.0.0 in libAndroid, libiOSDevice32 and libiOSDevice64 folders.

Bcrypt error message with C++ Builder

If you encounter an error message with BCrypt functions and you use C++ Builder, you have to add the file bcrypt.lib to your project.

Windows XP Compilation

If you want to compile your application for Windows XP, you need to uncomment //{$DEFINE WINXP} in tmscrypto.inc file.

Import a certificate from a PFX file or export a PFX file from a certificate

These methods use openssl.exe and are only available for Windows platform. There are openssl.exe files in libWin32 and libWin64 folder for both platforms. Maybe you need to set the environment variable OPENSSL_CONF to the path to the openssl.cnf file. You can add the following line to your project to set this variable:

```
ShellExecute(0, 'open', PChar('set'), PChar('OPENSSL_CONF=' +
ExpandFileName(PathToOpenSSLConf + 'openssl.cnf')), nil, SW_SHOW);
```