TMS Cryptography Pack
DEVELOPERS GUIDE

v2.5.1, September 2017

Web: http://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 & C++Builder XE2, XE3, XE4, XE5, XE6, XE7, XE8, 10 Seattle, 10.1 Berlin, 10.2 Tokyo.

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, Android, OSX32, iOS32, iOS64

If you want to use TMS Cryptography Pack in Win64 target, 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 or Android, you must add the directory “libAndroid” or “libiOSDevice32” or “libiOSDevice64” in the Search Path of the Project options.

Online references

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

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

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

-TMS ALL-ACCESS: http://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
- ✓ EdDSA
- ✓ ECIES
- ✓ SALSA
- ✓ SHA-2
- ✓ SHA-3
- ✓ PBKDF 2
- ✓ Blake2B
- ✓ RIPEMD-160
- ✓ Argon2

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;
    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 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;
  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
- `procedure EncryptFileW(s, o: string);` to encrypt a file whose path is s and the encrypted file path is o
- `procedure DecryptFileW(s, o: string);` to decrypt a 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 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

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;
  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

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;
  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

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, OAEP has been developed as a padding scheme. Similarly, for the signature, the secure form is called PSS. OAEP and PSS are described in the PKCS#1 v2.2.
We will use in our algorithms, RSA keys of length 2048, 3072 or 4096 bits.
The RSA class is:

```
TRSAKeyLength = (kl2048, kl3072, kl4096);

TOpenSSLFileType = (opensslpub, opensslpriv, opensslcert, opensslencpriv);

TRSAEncSign = class(TTMSCryptBase)
  public
    Constructor Create(AOwner: TComponent); overload; override;
    Constructor Create; reintroduce; overload;
    Constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; hashF: THashFunction; hashSB: Integer;
      outputFormat: TConvertType; uni: TUnicode); reintroduce; overload;
    Constructor Create(keyLength: TRSAKeyLength; modulus: string;
      publicExp: string; privateExp: string; hashF: THashFunction;
      hashSB: Integer; outputFormat: TConvertType; uni: TUnicode);
      reintroduce; 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;
    procedure FromOpenSSLCert(filePath: string);
    procedure FromOpenSSLPublicKey(filePath: string);
    procedure FromOpenSSLPrivateKey(filePath: string);
    procedure FromOpenSSLEncPrivateKey(filePath, Password: string);
    function OpenSSLTypeFile(filePath: string): TOpenSSLFileType;
  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: THashFunction read FHashFunction write FHashFunction
      default hsha2;
    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 passwd: string read Fpw;
    property withOpenSSL: boolean read FwithOpenSSL write SetwithOpenSSL 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: 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
- `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)
- `function OpenSSLTypeFile(filePath: string): TOpenSSLFileType;` to give the type an OpenSSL 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: THashFunction read FHashFunction write FHashFunction;` to choose the hash function (sha2 or sha3) used to hash the message before signature
- `property hashSizeBits: Integer read FHashSizeBits write SetHashSizeBits;` to choose the length in bits of the hash function output
- `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

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;
  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 := hsha2;
  rsa.hashSizeBits := 256;
  signature:= rsa.Sign('test');
  rsa.Free;
end;
```

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

EdDSA and ECIES

EdDSA is a digital signature algorithm using Edwards elliptic curves. It has been developed by a team directed by Daniel J. Bernstein. Two algorithms are implemented in this library, EdDSA25519, whose public key is encoded on 256 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.
The ECC (Elliptic Curve Cryptography) class is:

```
TECCType = (cc25519,cc511187);
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;
  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

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 type of curve (25519 or 511187)
- `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 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;
```

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;
  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

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-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 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;
  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 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

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;
  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

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;
  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

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.

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;
  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

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;
  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

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.

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;
{$IF (defined(MSWINDOWS) or defined(MACOS)) and (not defined(IOS))}
    function CharToFormat(charstring: PAnsiChar; charlen: Integer): string;
    function FormatToChar(str: string): PAnsiChar;
    function UnicodeToPAnsiChar(str: string): PAnsiChar;
    function PAnsiCharFromUnicodeLength(str: string): Integer;
{$ELSE}
    function CharToFormat(charstring: PByte; charlen: Integer): string;
    function FormatToChar(str: string): PByte;
{$IFEND}
    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;

  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** `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

The property is:

- property AType: TConvertType read FType write Ftype; to read and write the type of the conversion: hexa or base64

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.

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.

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;
    procedure LoadValues;
  public
    constructor Create(const FileName: string; const Key: string); overload;
    constructor Create(const FileName: string; const Encoding: TEncoding;
      CaseSensitive: Boolean); 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;
```

Troubleshooting

There 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.
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

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.