Intel® Integrated Performance Primitives

Cryptography Guide

IPP 7.1

Andrzej Chrzęszczyk
Jakub Chrzęszczyk

September, 2012

Foreword

The aim of this document is to make the first steps in using the IPP cryptography library easier. For this purpose we have prepared sample codes to almost all primitives described in the [MANUAL].

A general description of algorithms implemented in IPP cryptographic library may be found in several documents freely available on the Internet, so we have decided to restrict our bibliography to such positions.

Since the best possible compilation method of IPP code depends on several factors, we usually propose the simplest compilation methods of our examples. The reader interested in optimal compilation is referred to an appropriate user guide in *http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-documentation/*.

Contents

Part I

Private key cryptography

Chapter 1

Block cipher modes of operation

A block cipher is a cryptosystem that divides a long plain text into smaller, fixed size strings called blocks and enciphers one block at a time. The size of blocks depends on the algorithm and can be for example:

- 64-bit – for ARCFive,
- 128-bit – for Rijndael128,
- 192-bit for Rijndael192,
- 256-bit for Rijndael256.

In IPP we have at our disposal the following modes of operation on blocks.

- **Electronic Code Book mode (ECB)**,
- **Cipher Block Chaining mode (CBC)**,
- **Cipher Feedback mode (CFB)**,
- **Output Feedback mode (OFB)**,
- **Counter mode (CTR)**.

The detailed description of these modes can be found in [SP800-38A]. To clarify briefly how they work, let us assume that $e_k()$ is a function performing specific block cipher of block size b with key k. Let the plain text $x_1, x_2, \ldots$ be the sequence of blocks of length b and let $y_1, y_2, \ldots$ denote the corresponding cipher text blocks.
In ECB mode the encryption and decryption is performed as follows:

ECB encryption: $y_i = e_k(x_i), \quad i = 1, 2, \ldots,$
ECB decryption: $x_i = e_k^{-1}(y_i), \quad i = 1, 2, \ldots.$

To make the encryption process non-deterministic one introduces an initial value IV, which is a nonce i.e a number used only once. Usually it is derived from a counter value that is known to both parties and is incremented every time a new session starts. Using IV and XOR operation $\oplus$ one can summarize the two next modes.

CBC encryption: $y_1 = e_k(x_1 \oplus IV),$
$y_i = e_k(x_i \oplus y_{i-1}), \ i \geq 2,$
CBC decryption: $x_1 = e_k^{-1}(y_1) \oplus IV,$
$x_i = e_k^{-1}(y_i) \oplus y_{i-1}, \ i \geq 2.$

CFB encryption: $y_1 = e_k(IV) \oplus x_1,$
$y_i = e_k(y_{i-1}) \oplus x_i), \ i \geq 2,$
CFB decryption: $x_1 = e_k(IV) \oplus y_1,$
$x_i = e_k(y_{i-1}) \oplus y_i, \ i \geq 2.$

In OFB mode a key stream $s_1, s_2, \ldots$ is generated in a block-wise fashion and next it is used in encryption and decryption.

OFB encryption: $s_1 = e_k(IV), \quad y_1 = s_1 \oplus x_1,$
$s_i = e(s_{i-1}), \quad y_i = s_i \oplus x_i, \ i \geq 2,$
OFB decryption: $s_1 = e_k(IV), \quad x_1 = s_1 \oplus y_1,$
$s_i = e_k(s_{i-1}), \quad x_i = s_i \oplus y_i, \ i \geq 2.$

In the CTR mode the additional input to the block cipher is a counter CTR_i which assumes a different value every time the block cipher computes a new stream block.

CTR encryption: $y_i = e_k(IV \| CTR_i) \oplus x_i, \ i \geq 1,$
CTR decryption: $x_i = e_k(IV \| CTR_i) \oplus y_i, \ i \geq 1.$

The symbol $\|$ denotes the concatenation of strings.

Chapter 2

Rijndael

AES (Advanced Encryption Standard) is the most popular successor of DES/TDES. Its detailed description can be found in [FIPS197]. It uses the Rijndael algorithm developed by Joan Daemen and Vincent Rijmen.
In Rijndael algorithm the block sizes and key sizes can be (independently) set to 128, 192 or 256 bits. We shall describe the corresponding versions in separate sections.

Remark. AES uses the special case of Rijndael algorithm with block size equal to 128 bits. The AES keys can have 128, 192 or 256 bits.

2.1 Rijndael128 context preparation

To start encrypting and decrypting using Rijndael algorithm with 128-bit block size we have to prepare an appropriate context. The function `Rijndael128GetSize` shows how much memory we need.

```
IppStatus ippsRijndael128GetSize(int* size)       //output:
                        // Rijndael128 context size in bytes
```

Using the obtained value we can allocate a sufficient amount of memory. Next we can use a key and one of the functions `Rijndael128Init`, `SafeRijndael128Init` to initialize that context.

```
IppStatus ippsRijndael128Init(const Ipp8u* key,    //input:key
                    IppsRijndaelKeyLength keylen,    //input:
                                                  //key length
                    IppsRijndael128Spec* ctx)       //output:
                            //initialized Rijndael128 context
```

There are three possible values of `IppsRijndaelKeyLength`: `IppsRijndaelKey128`, `IppsRijndaelKey192` and `IppsRijndaelKey256`.

One can also choose slightly slower but more secure implementation of the algorithm.

```
IppStatus ippsSafeRijndael128Init(const Ipp8u* key,//input:key
                    IppsRijndaelKeyLength keylen,    //input:
                                                  //key length
                    IppsRijndael128Spec* ctx)       //output:
                            //initialized Rijndael128 context
```

2.2 Using Rijndael128 in ECB mode

To encrypt and decrypt we have to choose an appropriate operation mode. In ECB mode the encryption procedure can be performed with the use of `Rijndael128EncryptECB`.

```
IppStatus ippsRijndael128EncryptECB(const Ipp8u* plain,//input:
                                                    //plain text
                    Ipp8u* ciph,         //output:cipher text
                    int plainLen,   //input:plain text length
                    const IppsRijndael128Spec* ctx,  //input:
                                        //Rijndael128 context
                    IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael128DecryptECB`.

```
IppStatus ippsRijndael128DecryptECB(const Ipp8u* ciph,//input:
                                                    //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsRijndael128Spec* ctx,  //input:
                                        //Rijndael128 context
                      IppsCPPadding padding)    //input:padding
```

2.3 Rijndael128-ECB example

```
// Rijndael128 ECB encryption/decryption
// icpc -xHOST  -ipp=crypto  000AES00.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;           //block size
int ctxSize;                      //context size
ippsRijndael128GetSize(&ctxSize); //evaluating context size
// allocate memory for Rijndael context
IppsRijndael128Spec* ctx = (IppsRijndael128Spec*)
                                      ( new Ipp8u [ctxSize] );
// 128-bit key
//Ipp8u key[16] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
//0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

// 256-bit key
Ipp8u key[32] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// Rijndael context initialization
//ippsRijndael128Init(key,IppsRijndaelKey128,ctx);//128-bit key
                                                    //case
ippsRijndael128Init(key,IppsRijndaelKey256,ctx);  //256-bit key
// plain text
Ipp8u plain[] = "Rijndael-128 Electronic Code Book mode         ";
//cout<< sizeof(plain)<<endl;
```

```
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsRijndael128EncryptECB(plain,ciph,sizeof(plain),
ctx,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael128DecryptECB(ciph,deciph,sizeof(ciph),
ctx,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-128 Electronic Code Book mode
//Decrypted text: Rijndael-128 Electronic Code Book mode
```

2.4 Using Rijndael128 in CBC mode

In CBC mode the encryption procedure can be performed with the use of `Rijndael128EncryptCBC`.

```
IppStatus ippsRijndael128EncryptCBC(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsRijndael128Spec* ctx,  //input:
                                          //Rijndael128 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael128DecryptCBC`.

```
IppStatus ippsRijndael128DecryptCBC(const Ipp8u* ciph,//input:
                                                    //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsRijndael128Spec* ctx,  //input:
                                        //Rijndael128 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

2.5 Rijndael128-CBC example

```
// Rijndael128 CBC encryption/decryption
// icpc -xHOST  -ipp=crypto  000AES01.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;           // block size
int ctxSize;                      // context size
ippsRijndael128GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael128Spec* ctx = (IppsRijndael128Spec*)
                                     ( new Ipp8u [ctxSize] );
// 128-bit key
//Ipp8u key[16] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
//0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

// 256-bit key
Ipp8u key[32] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// Rijndael context initialization
//ippsRijndael128Init(key,IppsRijndaelKey128,ctx);//128-bit key
                                                    //case
ippsRijndael128Init(key,IppsRijndaelKey256,ctx);  //256-bit key
// plain text
Ipp8u plain[] = "Rijndael-128 Cipher Block Chaining mode        ";
```

```
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial key
Ipp8u iv[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
// encrypting plain text
ippsRijndael128EncryptCBC(plain,ciph,sizeof(plain),
ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael128DecryptCBC(ciph,deciph,sizeof(ciph),
ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-128 Cipher Block Chaining mode
//Decrypted text: Rijndael-128 Cipher Block Chaining mode
```

2.6 Using Rijndael128 in CFB mode

In CFB mode the encryption procedure can be performed with the use of `Rijndael128EncryptCFB`.

```
IppStatus ippsRijndael128EncryptCFB(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,          //output:cipher text
                      int plainLen,   //input:plain text length
                      int blkSize,           //input:block size
                      const IppsRijndael128Spec* ctx,  //input:
                                          //Rijndael128 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael128DecryptCFB`.

```
IppStatus ippsRijndael128DecryptCFB(const Ipp8u* ciph,//input:
                                                    //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsRijndael128Spec* ctx,  //input:
                                        //Rijndael128 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

2.7 Rijndael128-CFB example

```
// Rijndael128 CFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000AES02.cpp
#include "ippcp.h"
//#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;           // block size
int ctxSize;                      // context size
ippsRijndael128GetSize(&ctxSize); // evaluating  context size
// allocate memory for Rijndael context
IppsRijndael128Spec* ctx = (IppsRijndael128Spec*)
                                        ( new Ipp8u [ctxSize] );
// 128-bit key
//Ipp8u key[16] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
//0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

// 256-bit key
Ipp8u key[32] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// Rijndael context initialization
//ippsRijndael128Init(key,IppsRijndaelKey128,ctx);//128-bit key
                                                    //case
```

```
ippsRijndael128Init(key,IppsRijndaelKey256,ctx);  //256-bit key
// plain text
Ipp8u plain[] = "Rijndael-128 Cipher Feedback mode of operation ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
// encrypting plain text
ippsRijndael128EncryptCFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael128DecryptCFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-128 Cipher Feedback mode of operation
//Decrypted text: Rijndael-128 Cipher Feedback mode of operation
```

2.8 Using Rijndael128 in OFB mode

In OFB mode the encryption procedure can be performed with the use of `Rijndael128EncryptOFB`.

```
IppStatus ippsRijndael128EncryptOFB(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsRijndael128Spec* ctx,  //input:
                                         //Rijndael128 context
                     const Ipp8u* iv)   //input:initial vector
```

In the decryption we can use `Rijndael128DecryptOFB`.

```
IppStatus ippsRijndael128DecryptOFB(const Ipp8u* ciph,//input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsRijndael128Spec* ctx,  //input:
                                         //Rijndael128 context
                     const Ipp8u* iv)   //input:initial vector
```

2.9 Rijndael128-OFB example

```
// Rijndael128 OFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000AES03.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;           // block size
int ctxSize;                      // context size
ippsRijndael128GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael128Spec* ctx = (IppsRijndael128Spec*)
                                        ( new Ipp8u [ctxSize] );
// 128-bit key
```

```
//Ipp8u key[16] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
//0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

// 256-bit key
Ipp8u key[32] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// Rijndael context initialization
//ippsRijndael128Init(key,IppsRijndaelKey128,ctx);//128-bit key
                                                  //case
ippsRijndael128Init(key,IppsRijndaelKey256,ctx);  //256-bit key
// plain text
Ipp8u plain[]="Rijndael-128 Output Feedback mode of operation ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv0[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
Ipp8u iv[blkSize];
memcpy(iv,iv0,sizeof(iv0));
// encrypting plain text
ippsRijndael128EncryptOFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv0);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael128DecryptOFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-128 Output Feedback mode of operation
//Decrypted text: Rijndael-128 Output Feedback mode of operation
```

2.10 Using Rijndael128 in CTR mode

In CTR mode the encryption procedure can be performed with the use of `Rijndael128EncryptCTR`.

```
IppStatus ippsRijndael128EncryptCTR(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,          //output:cipher text
                      int plainLen,   //input:plain text length
                      const IppsRijndael128Spec* ctx,  //input:
                                          //Rijndael128 context
                      const Ipp8u* ctr,          //input:counter
                      int ctrNumBitSize)   //input:counter size
```

In the decryption we can use `Rijndael128DecryptCTR`.

```
IppStatus ippsRijndael128DecryptCTR(const Ipp8u* ciph,//input:
                                                   //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsRijndael128Spec* ctx,  //input:
                                          //Rijndael128 context
                      const Ipp8u* ctr,          //input:counter
                      int ctrNumBitSize)   //input:counter size
```

2.11 Rijndael128-CTR example

```
// Rijndael128 CTR encryption/decryption
// icpc -xHOST  -ipp=crypto  000AES04.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;           // block size
int ctxSize;                      // context size
ippsRijndael128GetSize(&ctxSize); // evaluating  context size
```

```
// allocate memory for Rijndael context
IppsRijndael128Spec* ctx = (IppsRijndael128Spec*)
                                        ( new Ipp8u [ctxSize] );
// 128-bit key
//Ipp8u key[16] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
//0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};

// 256-bit key
Ipp8u key[32] = {0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// Rijndael context initialization
//ippsRijndael128Init(key,IppsRijndaelKey128,ctx);//128-bit
                                                    //case
ippsRijndael128Init(key,IppsRijndaelKey256,ctx);  //256-bit key
// plain text
Ipp8u plain[]="Rijndael-128 Counter mode of operation";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to plain text size
Ipp8u ciph[sizeof(plain)];
// counter
Ipp8u ctr0[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
Ipp8u ctr[blkSize];
memcpy(ctr,ctr0,sizeof(ctr0));
int ctrNumBitSize = 64;
// encrypting plain text
ippsRijndael128EncryptCTR(plain,ciph,sizeof(plain),
ctx,ctr,ctrNumBitSize);
Ipp8u deciph[sizeof(ciph)];
memcpy(ctr,ctr0,sizeof(ctr0));
// decrypting cipher text
ippsRijndael128DecryptCTR(ciph,deciph,sizeof(ciph),
ctx,ctr,ctrNumBitSize);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
```

```
return 0;}

//Output:
//Plain text: Rijndael-128 Counter mode of operation
//Decrypted text: Rijndael-128 Counter mode of operation
```

2.12 Rijndael192 context preparation

To start encrypting and decrypting using Rijndael algorithm with 192-bit block size we have to prepare an appropriate context. The function `Rijndael192GetSize` shows how much memory we need.

```
 IppStatus ippsRijndael192GetSize(int* size)       //output:
                         // Rijndael192 context size in bytes
```

Using the obtained value we can allocate a sufficient amount of memory. Next we can use a 192-bit key and the function `Rijndael192Init` to initialize that context.

```
IppStatus ippsRijndael192Init(const Ipp8u* key,    //input:key
                      IppsRijndaelKeyLength keylen,    //input:
                                                 //key length
                      IppsRijndael192Spec* ctx)       //output:
                              //initialized Rijndael192 context
```

2.13 Using Rijndael192 in ECB mode

To encrypt and decrypt we have to choose an appropriate operation mode. In ECB mode the encryption procedure can be performed with the use of `Rijndael192EncryptECB`.

```
IppStatus ippsRijndael192EncryptECB(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,         //output:cipher text
                      int plainLen,   //input:plain text length
                      const IppsRijndael192Spec* ctx,  //input:
                                          //Rijndael192 context
                      IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael192DecryptECB`.

```
IppStatus ippsRijndael192DecryptECB(const Ipp8u* ciph,//input:
                                                    //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsRijndael192Spec* ctx,  //input:
                                        //Rijndael192 context
                      IppsCPPadding padding)    //input:padding
```

2.14 Rijndael192-ECB example

```
// Rijndael192 ECB encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael1920.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 24;           // block size
int ctxSize;                      // context size
ippsRijndael192GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael192Spec* ctx = (IppsRijndael192Spec*)
                                     ( new Ipp8u [ctxSize] );
// 192-bit key
Ipp8u key[24]={0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
               0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
               0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
// Rijndael context initialization
ippsRijndael192Init(key,IppsRijndaelKey192,ctx);
// plain text
Ipp8u plain[]="Rijndael-192 Electronic Code Book mode         ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array
Ipp8u ciph[sizeof(plain)];
//cout<< sizeof(ciph)<<endl;
// encrypting plain text
```

```
ippsRijndael192EncryptECB(plain,ciph,sizeof(plain),
ctx,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael192DecryptECB(ciph,deciph,sizeof(ciph),
ctx,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}
//Output:
//Plain text: Rijndael-192 Electronic Code Book mode
//Decrypted text: Rijndael-192 Electronic Code Book mode
```

2.15 Using Rijndael192 in CBC mode

In CBC mode the encryption procedure can be performed with the use of `Rijndael192EncryptCBC`.

```
IppStatus ippsRijndael192EncryptCBC(const Ipp8u* plain,//input:
                                                    //plain text
                    Ipp8u* ciph,          //output:cipher text
                    int plainLen,   //input:plain text length
                    const IppsRijndael192Spec* ctx,  //input:
                                        //Rijndael192 context
                    const Ipp8u* iv,   //input:initial vector
                    IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael192DecryptCBC`.

```
IppStatus ippsRijndael192DecryptCBC(const Ipp8u* ciph,//input:
                                                   //cipher text
                    Ipp8u* decr,      //output:decrypted text
                    int ciphLen,   //input:cipher text length
                    const IppsRijndael192Spec* ctx,  //input:
                                        //Rijndael192 context
                    const Ipp8u* iv,   //input:initial vector
                    IppsCPPadding padding)    //input:padding
```

2.16 Rijndael192-CBC example

```
// Rijndael192 CBC encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael1921.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 24;           // block size
int ctxSize;                      // context size
ippsRijndael192GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael192Spec* ctx = (IppsRijndael192Spec*)
                                       ( new Ipp8u [ctxSize] );
// key
Ipp8u key[24]={0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
               0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
               0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
// Rijndael context initialization
ippsRijndael192Init(key,IppsRijndaelKey192,ctx);
// plain text
Ipp8u plain[]="Rijndael-192 Cipher Block Chaining mode        ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text
Ipp8u ciph[sizeof(plain)];
// initial vector
Ipp8u iv[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                     0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
// encrypting plain text
ippsRijndael192EncryptCBC(plain,ciph,sizeof(plain),
ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael192DecryptCBC(ciph,deciph,sizeof(ciph),
ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
```

```
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-192 Cipher Block Chaining mode
//Decrypted text: Rijndael-192 Cipher Block Chaining mode
```

2.17 Using Rijndael192 in CFB mode

In CFB mode the encryption procedure can be performed with the use of `Rijndael192EncryptCFB`.

```
IppStatus ippsRijndael192EncryptCFB(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,          //output:cipher text
                      int plainLen,   //input:plain text length
                      int blkSize,            //input:block size
                      const IppsRijndael192Spec* ctx,  //input:
                                          //Rijndael192 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael192DecryptCFB`.

```
IppStatus ippsRijndael192DecryptCFB(const Ipp8u* ciph,//input:
                                                   //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      int blkSize,            //input:block size
                      const IppsRijndael192Spec* ctx,  //input:
                                          //Rijndael192 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

2.18 Rijndael192-CFB example

```
// Rijndael192 CFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael1922.cpp
```

```
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 24;           // block size
int ctxSize;                      // context size
ippsRijndael192GetSize(&ctxSize); // evaluating  context size
// allocate memory for Rijndael context
IppsRijndael192Spec* ctx = (IppsRijndael192Spec*)
                                        ( new Ipp8u [ctxSize] );
// key
Ipp8u key[24]={0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
               0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
               0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
// Rijndael context initialization
ippsRijndael192Init(key,IppsRijndaelKey192,ctx);
// plain text
Ipp8u plain[]="Rijndael-192 Cipher Feedback mode of operation ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array
Ipp8u ciph[sizeof(plain)];
// initial vector
Ipp8u iv[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                     0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
// encrypting plain text
ippsRijndael192EncryptCFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael192DecryptCFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
```

```
//Plain text: Rijndael-192 Cipher Feedback mode of operation
//Decrypted text: Rijndael-192 Cipher Feedback mode of operation
```

2.19 Using Rijndael192 in OFB mode

In OFB mode the encryption procedure can be performed with the use of `Rijndael192EncryptOFB`.

```
IppStatus ippsRijndael192EncryptOFB(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsRijndael192Spec* ctx,  //input:
                                          //Rijndael192 context
                     const Ipp8u* iv)   //input:initial vector
```

In the decryption we can use `Rijndael192DecryptOFB`.

```
IppStatus ippsRijndael192DecryptOFB(const Ipp8u* ciph,//input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsRijndael192Spec* ctx,  //input:
                                          //Rijndael192 context
                     const Ipp8u* iv)   //input:initial vector
```

2.20 Rijndael192-OFB example

```
// Rijndael192 OFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael1923.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
```

```
int main(){
const int blkSize = 24;           // block size
int ctxSize;                      // context size
ippsRijndael192GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael192Spec* ctx = (IppsRijndael192Spec*)
                                        ( new Ipp8u [ctxSize] );
// key
Ipp8u key[24]={0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
               0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
               0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
// Rijndael context initialization
ippsRijndael192Init(key,IppsRijndaelKey192,ctx);
// plain text
Ipp8u plain[]="Rijndael-192 Output Feedback mode of operation ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array
Ipp8u ciph[sizeof(plain)];
// initial vector
Ipp8u iv0[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                      0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                      0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
Ipp8u iv[blkSize];
memcpy(iv,iv0,sizeof(iv0));
// encrypting plain text
ippsRijndael192EncryptOFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv0);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael192DecryptOFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-192 Output Feedback mode of operation
```

```
//Decrypted text: Rijndael-192 Output Feedback mode of operation
```

2.21 Using Rijndael192 in CTR mode

In CTR mode the encryption procedure can be performed with the use of `Rijndael192EncryptCTR`.

```
IppStatus ippsRijndael192EncryptCTR(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,         //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsRijndael192Spec* ctx,  //input:
                                        //Rijndael192 context
                     const Ipp8u* ctr,         //input:counter
                     int ctrNumBitSize)   //input:counter size
```

In the decryption we can use `Rijndael192DecryptCTR`.

```
IppStatus ippsRijndael192DecryptCTR(const Ipp8u* ciph,//input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsRijndael192Spec* ctx,  //input:
                                        //Rijndael192 context
                     const Ipp8u* ctr,         //input:counter
                     int ctrNumBitSize)   //input:counter size
```

2.22 Rijndael192-CTR example

```
// Rijndael192 CTR encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael1924.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
```

```
const int blkSize = 24;           // block size
int ctxSize;                      // context size
ippsRijndael192GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael192Spec* ctx = (IppsRijndael192Spec*)
                                    ( new Ipp8u [ctxSize] );
// key
Ipp8u key[24]={0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
               0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
               0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
// Rijndael context initialization
ippsRijndael192Init(key,IppsRijndaelKey192,ctx);
// plain text
Ipp8u plain[]="Rijndael-192 Counter mode of operation         ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to plain text size
Ipp8u ciph[sizeof(plain)];
// counter array
Ipp8u ctr0[blkSize] ={0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                      0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                      0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
Ipp8u ctr[blkSize];
memcpy(ctr,ctr0,sizeof(ctr0));
int ctrNumBitSize = 64;
// encrypting plain text
ippsRijndael192EncryptCTR(plain,ciph,sizeof(plain),
ctx,ctr,ctrNumBitSize);
Ipp8u deciph[sizeof(ciph)];
memcpy(ctr,ctr0,sizeof(ctr0));
// decrypting cipher text
ippsRijndael192DecryptCTR(ciph,deciph,sizeof(ciph),
ctx,ctr,ctrNumBitSize);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
```

```
//Plain text: Rijndael-192 Counter mode of operation
//Decrypted text: Rijndael-192 Counter mode of operation
```

2.23 Rijndael256 context preparation

To start encrypting and decrypting using Rijndael algorithm with 256-bit block size we have to prepare an appropriate context. The function `Rijndael256GetSize` shows how much memory we need.

```
 IppStatus ippsRijndael256GetSize(int* size)       //output:
                        // Rijndael256 context size in bytes
```

Using the obtained value we can allocate a sufficient amount of memory. Next we can use a 256-bit key and the function `Rijndael256Init` to initialize that context.

```
IppStatus ippsRijndael256Init(const Ipp8u* key,    //input:key
                     IppsRijndaelKeyLength keylen,    //input:
                                                 //key length
                     IppsRijndael256Spec* ctx)       //output:
                             //initialized Rijndael256 context
```

2.24 Using Rijndael256 in ECB mode

To encrypt and decrypt we have to choose an appropriate operation mode. In ECB mode the encryption procedure can be performed with the use of `Rijndael256EncryptECB`.

```
IppStatus ippsRijndael256EncryptECB(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,         //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsRijndael256Spec* ctx,  //input:
                                        //Rijndael256 context
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael256DecryptECB`.

```
IppStatus ippsRijndael256DecryptECB(const Ipp8u* ciph,//input:
                                                    //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                      IppsCPPadding padding)    //input:padding
```

2.25 Rijndael256-ECB example

```
// Rijndael256 ECB encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael2560.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 32;           // block size
int ctxSize;                      // context size
ippsRijndael256GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael256Spec* ctx = (IppsRijndael256Spec*)
                                     ( new Ipp8u [ctxSize] );
// key
Ipp8u key[32]={0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
               0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
               0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
               0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
// Rijndael context initialization
ippsRijndael256Init(key,IppsRijndaelKey256,ctx);
// plain text
Ipp8u plain[]="Rijndael-256 Electronic Code "
             "Book mode of operation            ";
cout << "Plain text: "<< endl<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsRijndael256EncryptECB(plain,ciph,sizeof(plain),
ctx,IppsCPPaddingNONE);
```

```
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael256DecryptECB(ciph,deciph,sizeof(ciph),
ctx,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< endl << deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text:
//Rijndael-256 Electronic Code Book mode of operation
//Decrypted text:
//Rijndael-256 Electronic Code Book mode of operation
```

2.26 Using Rijndael256 in CBC mode

In CBC mode the encryption procedure can be performed with the use of `Rijndael256EncryptCBC`.

```
IppStatus ippsRijndael256EncryptCBC(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,          //output:cipher text
                      int plainLen,   //input:plain text length
                      const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael256DecryptCBC`.

```
IppStatus ippsRijndael256DecryptCBC(const Ipp8u* ciph,//input:
                                                   //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

2.27 Rijndael256-CBC example

```
// Rijndael256 CBC encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael2561.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 32;           // block size
int ctxSize;                      // context size
ippsRijndael256GetSize(&ctxSize); // evaluating  context size
// allocate memory for Rijndael context
IppsRijndael256Spec* ctx = (IppsRijndael256Spec*)
                                      ( new Ipp8u [ctxSize] );
// key
Ipp8u key[32]={0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
               0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
               0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
               0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
// Rijndael context initialization
ippsRijndael256Init(key,IppsRijndaelKey256,ctx);
// plain text
Ipp8u plain[]="Rijndael-256 Cipher Block "
              "Chaining mode of operation          ";
cout << "Plain text: "<< endl << plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                     0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
// encrypting plain text
ippsRijndael256EncryptCBC(plain,ciph,sizeof(plain),
ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael256DecryptCBC(ciph,deciph,sizeof(ciph),
```

```
ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< endl << deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text:
//Rijndael-256 Cipher Block Chaining mode of operation
//Decrypted text:
//Rijndael-256 Cipher Block Chaining mode of operation
```

2.28 Using Rijndael256 in CFB mode

In CFB mode the encryption procedure can be performed with the use of `Rijndael256EncryptCFB`.

```
IppStatus ippsRijndael256EncryptCFB(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,          //output:cipher text
                      int plainLen,   //input:plain text length
                      int blkSize,           //input:block size
                      const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

In the decryption we can use `Rijndael256DecryptCFB`.

```
IppStatus ippsRijndael256DecryptCFB(const Ipp8u* ciph,//input:
                                                   //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      int blkSize,           //input:block size
                      const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                      const Ipp8u* iv,   //input:initial vector
                      IppsCPPadding padding)    //input:padding
```

2.29 Rijndael256-CFB example

```
// Rijndael256 CFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael2562.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 32;           // block size
int ctxSize;                      // context size
ippsRijndael256GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael256Spec* ctx = (IppsRijndael256Spec*)
                                        ( new Ipp8u [ctxSize] );
// key
Ipp8u key[32]={0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
               0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
               0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
               0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
// Rijndael context initialization
ippsRijndael256Init(key,IppsRijndaelKey256,ctx);
// plain text
Ipp8u plain[]="Rijndael-256 Cipher Feedback "
              "mode of operation                ";
cout << "Plain text: "<< endl <<  plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                     0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
// encrypting plain text
ippsRijndael256EncryptCFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael256DecryptCFB(ciph,deciph,sizeof(ciph),
```

```
blkSize,ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< endl << deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text:
//Rijndael-256 Cipher Feedback mode of operation
//Decrypted text:
//Rijndael-256 Cipher Feedback mode of operation
```

2.30 Using Rijndael256 in OFB mode

In OFB mode the encryption procedure can be performed with the use of `Rijndael256EncryptOFB`.

```
IppStatus ippsRijndael256EncryptOFB(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                     const Ipp8u* iv)   //input:initial vector
```

In the decryption we can use `Rijndael256DecryptOFB`.

```
IppStatus ippsRijndael256DecryptOFB(const Ipp8u* ciph,//input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                     const Ipp8u* iv)   //input:initial vector
```

2.31 Rijndael256-OFB example

```
// Rijndael256 OFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael2563.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 32;           // block size
int ctxSize;                      // context size
ippsRijndael256GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael256Spec* ctx = (IppsRijndael256Spec*)
                                   ( new Ipp8u [ctxSize] );
// key
Ipp8u key[32]={0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
               0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
               0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
               0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
// Rijndael context initialization
ippsRijndael256Init(key,IppsRijndaelKey256,ctx);
// plain text
Ipp8u plain[]="Rijndael-256 Output Feedback "
             "mode of operation                ";
//cout<< sizeof(plain)<<endl;
cout << "Plain text: "<< endl << plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv0[blkSize] = {0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                      0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                      0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                      0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
Ipp8u iv[blkSize];
memcpy(iv,iv0,sizeof(iv0));
// encrypting plain text
ippsRijndael256EncryptOFB(plain,ciph,sizeof(plain),
```

```
blkSize,ctx,iv0);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsRijndael256DecryptOFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv);
cout <<"Decrypted text: "<< endl <<  deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text:
//Rijndael-256 Output Feedback mode of operation
//Decrypted text:
//Rijndael-256 Output Feedback mode of operation
```

2.32 Using Rijndael256 in CTR mode

In CTR mode the encryption procedure can be performed with the use of `Rijndael256EncryptCTR`.

```
IppStatus ippsRijndael256EncryptCTR(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,          //output:cipher text
                      int plainLen,   //input:plain text length
                      const IppsRijndael256Spec* ctx,  //input:
                                          //Rijndael256 context
                      const Ipp8u* ctr,          //input:counter
                      int ctrNumBitSize)   //input:counter size
```

In the decryption we can use `Rijndael256DecryptCTR`.

```
IppStatus ippsRijndael256DecryptCTR(const Ipp8u* ciph,//input:
                                                    //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsRijndael256Spec* ctx,  //input:
                                        //Rijndael256 context
                     const Ipp8u* ctr,         //input:counter
                     int ctrNumBitSize)   //input:counter size
```

2.33 Rijndael256-CTR example

```
// Rijndael256 CTR encryption/decryption
// icpc -xHOST  -ipp=crypto  000Rijndael2564.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 32;           // block size
int ctxSize;                      // context size
ippsRijndael256GetSize(&ctxSize); // evaluating context size
// allocate memory for Rijndael context
IppsRijndael256Spec* ctx = (IppsRijndael256Spec*)
                                    ( new Ipp8u [ctxSize] );
// key
Ipp8u key[32]={0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
               0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
               0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
               0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
// Rijndael context initialization
ippsRijndael256Init(key,IppsRijndaelKey256,ctx);
// plain text
Ipp8u plain[]="Rijndael-256 Counter mode      ";
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to plain text size
Ipp8u ciph[sizeof(plain)];
// counter
```

```
Ipp8u ctr0[blkSize]={0x0f,0x0e,0x0d,0x0c,0x0b,0x0a,0x09,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,
                     0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                     0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00};
Ipp8u ctr[blkSize];
memcpy(ctr,ctr0,sizeof(ctr0));
int ctrNumBitSize = 64;
// encrypting plain text
ippsRijndael256EncryptCTR(plain,ciph,sizeof(plain),
ctx,ctr,ctrNumBitSize);
Ipp8u deciph[sizeof(ciph)];
memcpy(ctr,ctr0,sizeof(ctr0));
// decrypting cipher text
ippsRijndael256DecryptCTR(ciph,deciph,sizeof(ciph),
ctx,ctr,ctrNumBitSize);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: Rijndael-256 Counter mode
//Decrypted text: Rijndael-256 Counter mode
```

Chapter 3

ARCFive

The ARCFive cipher is a IPP version of Ron Rivest's RC5, described in [RC5].

3.1 ARCFive64 context preparation

To start encrypting and decrypting with ARCFive64 we have to prepare an appropriate context. The function `ARCFive64GetSize` shows how much memory we need.

```
IppStatus ippsARCFive64GetSize(int rounds,    //input:number
                                                 //of rounds
                        int* size)//output:ARCFive64 context
                                                //size in bytes
```

Using the obtained value we can allocate a sufficient amount of memory. Next we can use the function `ARCFive64Init` to initialize that context.

```
IppStatus ippsARCFive64Init(const Ipp8u* key,      //input:key
                      int keyLen,            //input:key length
                      int rounds,      //input:number of rounds
                      IppsARCFive64Spec* ctx)         //output:
                                 //initialized ARCFive64 context
```

3.2 Using ARCFive64 in ECB mode

To encrypt and decrypt we have to choose an appropriate operation mode. In ECB mode the encryption procedure can be performed with the use of

`ARCFive64EncryptECB`.

```
IppStatus ippsARCFive64EncryptECB(const Ipp8u* plain, //input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsARCFive64Spec* ctx,    //input:
                                        //ARCFive64 context
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `ARCFive64DecryptECB`.

```
IppStatus ippsARCFive64DecryptECB(const Ipp8u* ciph,  //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsARCFive64Spec* ctx,    //input:
                                        //ARCFive64 context
                     IppsCPPadding padding)    //input:padding
```

3.3 ARCFive64-ECB example

```
// ARCFive64 ECB encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive640.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 8;      // ARCFive64 block size
int rounds=18;
int ctxSize;                // ARCFive64 context size
// evaluating ARCFive64 context size
ippsARCFive64GetSize(rounds,&ctxSize);
// allocate memory for ARCFive64 context
IppsARCFive64Spec* ctx = (IppsARCFive64Spec*)
                                    ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive64 context initialization
```

```
ippsARCFive64Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]="ARCFive64 Electronic Code Book mode            ";
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsARCFive64EncryptECB(plain,ciph,sizeof(plain),
ctx,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive64DecryptECB(ciph,deciph,sizeof(ciph),
ctx,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive64 Electronic Code Book mode
//Decrypted text: ARCFive64 Electronic Code Book mode
```

3.4 Using ARCFive64 in CBC mode

In CBC mode an additional component – the initial vector appears. The encryption procedure can be performed with the use of `ARCFive64EncryptCBC`.

```
IppStatus ippsARCFive64EncryptCBC(const Ipp8u* plain, //input:
                                                  //plain text
                     Ipp8u* ciph,         //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsARCFive64Spec* ctx,    //input:
                                       //ARCFive64 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `ARCFive64DecryptCBC`.

```
IppStatus ippsARCFive64DecryptCBC(const Ipp8u* ciph,  //input:
                                                    //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsARCFive64Spec* ctx,    //input:
                                          //ARCFive64 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

3.5 ARCFive64-CBC example

```
// ARCFive64 CBC encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive641.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 8;       // ARCFive64 block size
int rounds=18;
int ctxSize;                 // ARCFive64 context size
// evaluating ARCFive64 context size
ippsARCFive64GetSize(rounds,&ctxSize);
// allocate memory for ARCFive64 context
IppsARCFive64Spec* ctx = (IppsARCFive64Spec*)
                                  ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive64 context initialization
ippsARCFive64Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]={"ARCFive64 Cipher Block Chaining mode           "};
cout << "Plain text: "<< plain << endl;
// initial vector
Ipp8u iv[blkSize] = {0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsARCFive64EncryptCBC(plain,ciph,sizeof(plain),
ctx,iv,IppsCPPaddingNONE);
```

```
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive64DecryptCBC(ciph,deciph,sizeof(ciph),
ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive64 Cipher Block Chaining mode
//Decrypted text: ARCFive64 Cipher Block Chaining mode
```

3.6 Using ARCFive64 in CFB mode

The encryption procedure can be performed using `ARCFive64EncryptCFB`.

```
IppStatus ippsARCFive64EncryptCFB(const Ipp8u* plain, //input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsARCFive64Spec* ctx,    //input:
                                          //ARCFive64 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `ARCFive64DecryptCFB`.

```
IppStatus ippsARCFive64DecryptCFB(const Ipp8u* ciph,  //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsARCFive64Spec* ctx,    //input:
                                         //ARCFive64 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

3.7 ARCFive64-CFB example

```
// ARCFive64 CFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive642.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 8;       // ARCFive64 block size
int rounds=18;
int ctxSize;                 // ARCFive64 context size
// evaluating ARCFive64 context size
ippsARCFive64GetSize(rounds,&ctxSize);
// allocate memory for ARCFive64 context
IppsARCFive64Spec* ctx = (IppsARCFive64Spec*)
                                    ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive64 context initialization
ippsARCFive64Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]=
{"ARCFive64 Cipher Feedback mode of operation    "};
cout << "Plain text: "<< plain << endl;
// initial vector
Ipp8u iv[blkSize]={0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsARCFive64EncryptCFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive64DecryptCFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}
```

```
//Output:
//Plain text: ARCFive64 Cipher Feedback mode of operation
//Decrypted text: ARCFive64 Cipher Feedback mode of operation
```

3.8 Using ARCFive64 in OFB mode

The encryption procedure can be performed using `ARCFive64EncryptOFB`.

```
IppStatus ippsARCFive64EncryptOFB(const Ipp8u* plain, //input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsARCFive64Spec* ctx,    //input:
                                          //ARCFive64 context
                     const Ipp8u* iv)   //input:initial vector
```

In the decryption we can use `ARCFive64DecryptOFB`.

```
IppStatus ippsARCFive64DecryptOFB(const Ipp8u* ciph,  //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsARCFive64Spec* ctx,    //input:
                                         //ARCFive64 context
                     const Ipp8u* iv)   //input:initial vector
```

3.9 ARCFive64-OFB example

```
// ARCFive64 OFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive643.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
```

```
int main(){
const int blkSize = 8;       // ARCFive64 block size
int rounds=18;
int ctxSize;                 // ARCFive64 context size
// evaluating ARCFive64 context size
ippsARCFive64GetSize(rounds,&ctxSize);
// allocate memory for ARCFive64 context
IppsARCFive64Spec* ctx = (IppsARCFive64Spec*)
                                    ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive64 context initialization
ippsARCFive64Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]=
{"ARCFive64 Output Feedback mode of operation    "};
cout << "Plain text: "<< plain << endl;
// initial vector
Ipp8u iv0[blkSize]={0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
Ipp8u iv[blkSize];
memcpy(iv,iv0,sizeof(iv0));
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsARCFive64EncryptOFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv0);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive64DecryptOFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive64 Output Feedback mode of operation
//Decrypted text: ARCFive64 Output Feedback mode of operation
```

3.10 Using ARCFive64 in CTR mode

The encryption procedure can be performed using `ARCFive64EncryptCTR`.

```
IppStatus ippsARCFive64EncryptCTR(const Ipp8u* plain, //input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsARCFive64Spec* ctx,    //input:
                                          //ARCFive64 context
                     Ipp8u* ctr,               //input:counter
                     int ctrNumBitSize)   //input:counter size
```

In the decryption we can use `ARCFive64DecryptCTR`.

```
IppStatus ippsARCFive64DecryptCTR(const Ipp8u* ciph,  //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsARCFive64Spec* ctx,    //input:
                                          //ARCFive64 context
                     Ipp8u* ctr,               //input:counter
                     int ctrNumBitSize)   //input:counter size
```

3.11 ARCFive64-CTR example

```
// ARCFive64 CTR encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive644.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 8;       // ARCFive64 block size
int rounds=18;
int ctxSize;                 // ARCFive64 context size
// evaluating ARCFive64 context size
```

```
ippsARCFive64GetSize(rounds,&ctxSize);
// allocate memory for ARCFive64 context
IppsARCFive64Spec* ctx = (IppsARCFive64Spec*)
                                    ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive64 context initialization
ippsARCFive64Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]={"ARCFive64 Counter mode of operation"};
cout << "Plain text: "<< plain << endl;
// counter array
Ipp8u ctr0[blkSize]={0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88};
Ipp8u ctr[blkSize];
memcpy(ctr,ctr0,sizeof(ctr0));
int ctrNumBitSize = 64;
// cipher text array
Ipp8u ciph[sizeof(plain)];
// encrypting plain text
ippsARCFive64EncryptCTR(plain,ciph,sizeof(plain),
ctx,ctr,ctrNumBitSize);
Ipp8u deciph[sizeof(ciph)];
memcpy(ctr, ctr0, sizeof(ctr0));
// decrypting cipher text
ippsARCFive64DecryptCTR(ciph,deciph,sizeof(ciph),
ctx,ctr,ctrNumBitSize);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive64 Counter mode of operation
//Decrypted text: ARCFive64 Counter mode of operation
```

3.12 ARCFive128 context preparation

To start encrypting and decrypting with ARCFive128 we have to prepare an appropriate context. The function `ARCFive128GetSize` shows how much memory we need.

```
IppStatus ippsARCFive128GetSize(int rounds,    //input:number
                                                  //of rounds
                        int* size)//output:ARCFive128 context
                                                 //size in bytes
```

Using the obtained value we can allocate a sufficient amount of memory. Next we can use the function `ARCFive128Init` to initialize that context.

```
IppStatus ippsARCFive128Init(const Ipp8u* key,     //input:key
                      int keyLen,            //input:key length
                      int rounds,      //input:number of rounds
                      IppsARCFive128Spec* ctx)        //output:
                               //initialized ARCFive128 context
```

3.13 Using ARCFive128 in ECB mode

To encrypt and decrypt we have to choose an appropriate operation mode. In ECB mode the encryption procedure can be performed with the use of `ARCFive128EncryptECB`.

```
IppStatus ippsARCFive128EncryptECB(const Ipp8u* plain,//input:
                                                    //plain text
                      Ipp8u* ciph,         //output:cipher text
                      int plainLen,   //input:plain text length
                      const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                      IppsCPPadding padding)    //input:padding
```

In the decryption we can use `ARCFive128DecryptECB`.

```
IppStatus ippsARCFive128DecryptECB(const Ipp8u* ciph, //input:
                                                   //cipher text
                      Ipp8u* decr,      //output:decrypted text
                      int ciphLen,   //input:cipher text length
                      const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                      IppsCPPadding padding)    //input:padding
```

3.14 ARCFive128-ECB example

```
// ARCFive128 ECB encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive1280.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;      // ARCFive128 block size
int rounds=18;
int ctxSize;                 // ARCFive128 context size
// evaluating ARCFive128 context size
ippsARCFive128GetSize(rounds,&ctxSize);
// allocate memory for ARCFive128 context
IppsARCFive128Spec* ctx = (IppsARCFive128Spec*)
                                  ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2,
               0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive128 context initialization
ippsARCFive128Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]="ARCFive128 Electronic Code Book mode           ";
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// encrypting plain text
ippsARCFive128EncryptECB(plain,ciph,sizeof(plain),
ctx,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive128DecryptECB(ciph,deciph,sizeof(ciph),
ctx,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive128 Electronic Code Book mode
```

```
//Decrypted text: ARCFive128 Electronic Code Book mode
```

3.15 Using ARCFive128 in CBC mode

In CBC mode an additional component – the initial vector appears. The encryption procedure can be performed with the use of `ARCFive128EncryptCBC`.

```
IppStatus ippsARCFive128EncryptCBC(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `ARCFive128DecryptCBC`.

```
IppStatus ippsARCFive128DecryptCBC(const Ipp8u* ciph, //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

3.16 ARCFive128-CBC example

```
// ARCFive128 CBC encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive1281.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;       // ARCFive128 block size
int rounds=18;
int ctxSize;                  // ARCFive128 context size
```

```
// evaluating ARCFive128 context size
ippsARCFive128GetSize(rounds,&ctxSize);
// allocate memory for ARCFive128 context
IppsARCFive128Spec* ctx = (IppsARCFive128Spec*)
                                    ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2,
               0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive128 context initialization
ippsARCFive128Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]={"ARCFive128 Cipher Block Chaining mode          "};
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv[blkSize] = {0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
                     0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// encrypting plain text
ippsARCFive128EncryptCBC(plain,ciph,sizeof(plain),
ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive128DecryptCBC(ciph,deciph,sizeof(ciph),
ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive128 Cipher Block Chaining mode
//Decrypted text: ARCFive128 Cipher Block Chaining mode
```

3.17 Using ARCFive128 in CFB mode

The encryption procedure can be performed using `ARCFive128EncryptCFB`.

```
IppStatus ippsARCFive128EncryptCFB(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

In the decryption we can use `ARCFive128DecryptCFB`.

```
IppStatus ippsARCFive128DecryptCFB(const Ipp8u* ciph, //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     const Ipp8u* iv,   //input:initial vector
                     IppsCPPadding padding)    //input:padding
```

3.18 ARCFive128-CFB example

```
// ARCFive128 CFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive1282.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;       // ARCFive128 block size
int rounds=18;
int ctxSize;                  // ARCFive128 context size
// evaluating ARCFive128 context size
ippsARCFive128GetSize(rounds,&ctxSize);
// allocate memory for ARCFive128 context
IppsARCFive128Spec* ctx = (IppsARCFive128Spec*)
                                    ( new Ipp8u [ctxSize] );
```

```
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2,
               0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive128 context initialization
ippsARCFive128Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]={"ARCFive128 Cipher Feedback mode of operation   "};
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv[blkSize] = {0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
                     0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
// encrypting plain text
ippsARCFive128EncryptCFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv,IppsCPPaddingNONE);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive128DecryptCFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv,IppsCPPaddingNONE);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive128 Cipher Feedback mode of operation
//Decrypted text: ARCFive128 Cipher Feedback mode of operation
```

3.19 Using ARCFive128 in OFB mode

The encryption procedure can be performed using `ARCFive128EncryptOFB`.

```
IppStatus ippsARCFive128EncryptOFB(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     int blkSize,           //input:block size
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     const Ipp8u* iv)   //input:initial vector
```

In the decryption we can use `ARCFive128DecryptOFB`.

```
IppStatus ippsARCFive128DecryptOFB(const Ipp8u* ciph, //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     int blkSize,           //input:block size
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     const Ipp8u* iv)   //input:initial vector
```

3.20 ARCFive128-OFB example

```
// ARCFive128 OFB encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive1283.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;       // ARCFive128 block size
int rounds=18;
int ctxSize;                  // ARCFive128 context size
// evaluating ARCFive128 context size
ippsARCFive128GetSize(rounds,&ctxSize);
// allocate memory for ARCFive128 context
IppsARCFive128Spec* ctx = (IppsARCFive128Spec*)
```

```
                                    ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2,
               0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive128 context initialization
ippsARCFive128Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]={"ARCFive128 Output Feedback mode of operation   "};
cout << "Plain text: "<< plain << endl;
// cipher text array of size equal to multiple of blkSize
Ipp8u ciph[(sizeof(plain)+blkSize-1) &~(blkSize-1)];
// initial vector
Ipp8u iv0[blkSize] = {0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00,
                      0x77,0x66,0x55,0x44,0x33,0x22,0x11,0x00};
Ipp8u iv[blkSize];
memcpy(iv,iv0,sizeof(iv0));

// encrypting plain text
ippsARCFive128EncryptOFB(plain,ciph,sizeof(plain),
blkSize,ctx,iv0);
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFive128DecryptOFB(ciph,deciph,sizeof(ciph),
blkSize,ctx,iv);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive128 Output Feedback mode of operation
//Decrypted text: ARCFive128 Output Feedback mode of operation
```

3.21 Using ARCFive128 in CTR mode

The encryption procedure can be performed using `ARCFive128EncryptCTR`.

```
IppStatus ippsARCFive128EncryptCTR(const Ipp8u* plain,//input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     Ipp8u* ctr,               //input:counter
                     int ctrNumBitSize)   //input:counter size
```

In the decryption we can use `ARCFive128DecryptCTR`.

```
IppStatus ippsARCFive128DecryptCTR(const Ipp8u* ciph, //input:
                                                     //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsARCFive128Spec* ctx,   //input:
                                          //ARCFive128 context
                     Ipp8u* ctr,               //input:counter
                     int ctrNumBitSize)   //input:counter size
```

3.22 ARCFive128-CTR example

```
// ARCFive128 CTR encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFive1284.cpp
#include "ippcp.h"
#include<cstring>
#include<iostream>
using namespace std;
int main(){
const int blkSize = 16;       // ARCFive128 block size
int rounds=18;
int ctxSize;                  // ARCFive128 context size
// evaluating ARCFive128 context size
ippsARCFive128GetSize(rounds,&ctxSize);
// allocate memory for ARCFive128 context
IppsARCFive128Spec* ctx = (IppsARCFive128Spec*)
```

```
                                        ( new Ipp8u [ctxSize] );
Ipp8u key[] = {0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2,
               0xa,0xb,0xc,0xd,0xe,0xf,0x1,0x2};  // key
// ARCFive128 context initialization
ippsARCFive128Init(key,sizeof(key),rounds,ctx);
// plain text
Ipp8u plain[]={"ARCFive128 Counter mode of operation"};
cout << "Plain text: "<< plain << endl;
// counter array
Ipp8u ctr0[blkSize]={0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88,
                     0xff,0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88};
Ipp8u ctr[blkSize];
memcpy(ctr,ctr0,sizeof(ctr0));
int ctrNumBitSize = 64;
// cipher text array
Ipp8u ciph[sizeof(plain)];
// encrypting plain text
ippsARCFive128EncryptCTR(plain,ciph,sizeof(plain),
ctx,ctr,ctrNumBitSize);
Ipp8u deciph[sizeof(ciph)];
memcpy(ctr,ctr0,sizeof(ctr0));
// decrypting cipher text
ippsARCFive128DecryptCTR(ciph,deciph,sizeof(ciph),
ctx,ctr,ctrNumBitSize);
cout <<"Decrypted text: "<< deciph << endl;
delete (Ipp8u*) ctx;
return 0;}

//Output:
//Plain text: ARCFive128 Counter mode of operation
//Decrypted text: ARCFive128 Counter mode of operation
```

Chapter 4

ARCFour

The ARCFour cipher has the same functionality as RC4, which is the property of RSA Security Inc. A description can be found in [ARCFOUR].

4.1 ARCFour context preparation

To start encrypting and decrypting using ARCFour stream cipher we have to prepare an appropriate context. The function `ARCFourGetSize` shows how much memory we need.

```
IppStatus ippsARCFourGetSize(int* size)     //output:ARCFour
                                          //context size in bytes
```

Using the obtained value we can allocate a sufficient amount of memory. Next we can use the function `ARCFourInit` to initialize that context.

```
IppStatus ippsARCFourInit(const Ipp8u* key,        //input:key
                          int keyLen,       //input:key length
                          IppsARCFourState* ctx)     //output:
                                 //initialized ARCFour context
```

4.2 Using ARCFour

ARCFour stream cipher operates in OFB mode. The encryption procedure can be performed with the use of `ARCFourEncrypt`.

```
IppStatus ippsARCFourEncrypt(const Ipp8u* plain,      //input:
                                                    //plain text
                     Ipp8u* ciph,          //output:cipher text
                     int plainLen,   //input:plain text length
                     const IppsARCFourState* ctx)     //input:
                                         //ARCFour context
```

In the decryption we can use `ARCFourDecryptECB`.

```
IppStatus ippsARCFourDecrypt(const Ipp8u* ciph,       //input:
                                                   //cipher text
                     Ipp8u* decr,      //output:decrypted text
                     int ciphLen,   //input:cipher text length
                     const IppsARCFourState* ctx)     //input:
                                         //ARCFour context
```

We have also the following two functions at our disposal:

```
IppStatus ippsARCFourCheckKey(const Ipp8u* key,    //input:key
                     int keyLen,            //input:key length
                     IppsBool* IsWeak)            //output:key
                                      // weakness check result
```

```
IppStatus ippsARCFourReset(IppsARCFourState* ctx)    //output:
                               //re-initialized ARCFour context
```

4.3 ARCFour example

```
// ARCFour encryption/decryption
// icpc -xHOST  -ipp=crypto  000ARCFour.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;

int main(){
int ctxSize;  // context size
ippsARCFourGetSize(&ctxSize); // evaluating context size
```

```
// preparing memory for context
IppsARCFourState* ctx = (IppsARCFourState*)
                                ( new Ipp8u [ctxSize] );
// key
Ipp8u key[] = {0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
// context initialization
ippsARCFourInit(key,sizeof(key),ctx);
// plain text
Ipp8u plain[]="ARCFour Stream Cipher   ";
cout<<"Plain text: "<<plain<<endl;
// cipher array
Ipp8u ciph[sizeof(plain)];
// encrypting plain text
ippsARCFourEncrypt(plain,ciph,sizeof(plain),ctx);
// resetting stream
ippsARCFourReset(ctx);
// deciphered array
Ipp8u deciph[sizeof(ciph)];
// decrypting cipher text
ippsARCFourDecrypt(ciph, deciph,sizeof(ciph),ctx);
cout<<"Decrypted text: "<<deciph<<endl;
delete (Ipp8u*) ctx;
return 0;
}

//Output:
//Plain text: ARCFour Stream Cipher
//Decrypted text: ARCFour Stream Cipher
```

Chapter 5

Hash functions for non-streaming messages

A detailed description of the hash functions can be found in [FIPS 180-2], [RFC 1321].
In the case of relatively short messages one can apply IPP hash functions to the entire message at once. We begin this chapter from presentation of hash primitives of this kind.

5.1 MD5MessageDigest

The `MD5MessageDigest` function computes a 128-bit digest value of a variable length message.

```
IppStatus ippsMD5MessageDigest(const Ipp8u* msg,//input:message
                          int msgLen,    //input:message length
                          Ipp8u* digest)//output:message digest
```

```
// MD5 message digest example
// icpc -xHOST  -ipp=crypto  000Hash00.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
Ipp8u msg[] = "abc";
cout<<"Message: "<<msg<<endl;
Ipp8u digest[16];
ippsMD5MessageDigest(msg, strlen((char*)msg), digest);
cout<<"MD5: ";
```

```
for(int k=0;k<16;k++){
cout.fill('0');           // don't print n
cout.width(2);            // instead of 0n
cout<<hex<<(int)digest[k];
}
cout<<endl;
return 0;
}

//Output:
//Message: abc
//MD5: 900150983cd24fb0d6963f7d28e17f72
```

5.2 SHA1MessageDigest

The `SHA1MessageDigest` function computes a 160-bit digest value of a variable length message.

```
IppStatus ippsSHA1MessageDigest(const Ipp8u* msg,     //input:
                                                      //message
                          int msgLen,    //input:message length
                          Ipp8u* digest)//output:message digest
```

```
// SHA1 message digest example
// icpc -xHOST  -ipp=crypto  000Hash01.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
Ipp8u msg[] = "abc";       // example from FIPS 180-2
cout<<"Message: "<<msg<<endl;
Ipp8u digest[20];
ippsSHA1MessageDigest(msg, strlen((char*)msg), digest);
cout<<"SHA1: ";
for(int k=0;k<20;k++){
cout.fill('0');           // don't print n
cout.width(2);            // instead of 0n
cout<<hex<<(int)digest[k];
```

```
}
cout<<endl;
return 0;
}

//Output:
//Message: abc
//SHA1: a9993e364706816aba3e25717850c26c9cd0d89d
```

5.3 SHA224MessageDigest

The `SHA224MessageDigest` function computes a 224-bit digest value of a variable length message.

```
IppStatus ippsSHA224MessageDigest(const Ipp8u* msg,   //input:
                                                      //message
                          int msgLen,    //input:message length
                          Ipp8u* digest)//output:message digest
```

```
// SHA224 message digest example
// icpc -xHOST  -ipp=crypto  000Hash02.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
Ipp8u msg[] = "abc";
cout<<"Message: "<<msg<<endl;
Ipp8u digest[28];
ippsSHA224MessageDigest(msg, strlen((char*)msg), digest);
cout<<"SHA224: ";
for(int k=0;k<28;k++){
cout.fill('0');           // don't print n
cout.width(2);            // instead of 0n
cout<<hex<<(int)digest[k];
}
cout<<endl;
```

```
return 0;
}

//Output:
//Message: abc
//SHA224: 23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7
```

5.4 SHA256MessageDigest

The `SHA256MessageDigest` function computes a 256-bit digest value of a variable length message.

```
IppStatus ippsSHA256MessageDigest(const Ipp8u* msg,   //input:
                                                      //message
                          int msgLen,    //input:message length
                          Ipp8u* digest)//output:message digest
```

```
// SHA256 message digest example
// icpc -xHOST  -ipp=crypto  000Hash03.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
Ipp8u msg[] = "abc";
cout<<"Message: "<<msg<<endl;
Ipp8u digest[32];
ippsSHA256MessageDigest(msg, strlen((char*)msg), digest);
cout<<"SHA256: ";
for(int k=0;k<32;k++){
cout.fill('0');           // don't print n
cout.width(2);            // instead of 0n
cout<<hex<<(int)digest[k];
}
cout<<endl;
return 0;
}
```

```
//Output:
//Message: abc
//SHA256: ba7816bf8f01cfea414140de5dae2223b00361a396177a9c
          b410ff61f20015ad
```

5.5 SHA384MessageDigest

The `SHA384MessageDigest` function computes a 384-bit digest value of a variable length message.

```
IppStatus ippsSHA384MessageDigest(const Ipp8u* msg,   //input:
                                                      //message
                          int msgLen,    //input:message length
                          Ipp8u* digest)//output:message digest
```

```
// SHA384 message digest example
// icpc -xHOST  -ipp=crypto  000Hash04.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
Ipp8u msg[] = "abc";
cout<<"Message: "<<msg<<endl;
Ipp8u digest[48];
ippsSHA384MessageDigest(msg, strlen((char*)msg), digest);
cout<<"SHA384: ";
for(int k=0;k<48;k++){
cout.fill('0');           // don't print n
cout.width(2);            // instead of 0n
cout<<hex<<(int)digest[k];
}
cout<<endl;
return 0;
}

//Output:
```

```
//SHA384: cb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b
          605a43ff5bed8086072ba1e7cc2358baeca134c825a7
```

5.6 SHA512MessageDigest

The `SHA512MessageDigest` function computes a 512-bit digest value of a variable length message.

```
IppStatus ippsSHA512MessageDigest(const Ipp8u* msg,   //input:
                                                      //message
                          int msgLen,    //input:message length
                          Ipp8u* digest)//output:message digest
```

```
// SHA512 message digest example
// icpc -xHOST  -ipp=crypto  000Hash05.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
Ipp8u msg[] = "abc";
cout<<"Message: "<<msg<<endl;
Ipp8u digest[64];
ippsSHA512MessageDigest(msg, strlen((char*)msg), digest);
cout<<"SHA512: ";
for(int k=0;k<64;k++){
cout.fill('0');           // don't print n
cout.width(2);            // instead of 0n
cout<<hex<<(int)digest[k];
}
cout<<endl;
return 0;
}

//Output:
//Message: abc
//SHA512: ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee
          64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce8
          0e2a9ac94fa54ca49f
```

Chapter 6

Hash functions for streaming messages

6.1 MD5 context preparation

To start MD5 digest computations for streaming messages we have to prepare an appropriate context. The function `MD5GetSize` shows the amount of memory we need.

```
IppStatus ippsMD5GetSize(int* size)     //output:MD5 context
                                            // size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `MD5Init`.

```
IppStatus ippsMD5Init(IppsMD5State* ctx)//output:initialized
                                              //MD5 context
```

6.2 Using MD5 primitives

The current input message can be digested with the use of `MD5Update`.

```
IppStatus ippsMD5Update(const Ipp8u* msg,     //input:message
                        int msgLen,    //input:message length
                        IppsMD5State* ctx)//input:MD5 context
                                    //output:updated context
```

The `MD5Duplicate` function duplicates the `IppsMD5State` context.

```
IppStatus ippsMD5Duplicate(IppsMD5State* SrcCtx,      //input:
                                              //source context
                           IppsMD5State* DstCtx)     //output:
                                             //destination context
```

The function `MD5Final` completes calculation of the digest.

```
IppStatus ippsMD5Final(Ipp8u* dgst,            //output:digest
                       IppsMD5State* ctx)  //input:MD5 context
```

A call to the function `MD5GetTag` retains the possibility to update the digest and computes the authentication tag.

```
IppStatus ippsMD5GetTag(Ipp8u* tag,//output:authentication tag
                        IppsMD5State* ctx) //input:MD5 context
```

6.3 MD5 example

```
// MD5 digest of a streaming message:
// icpc -xHOST  -ipp=crypto  000Hash10.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
int ctxSize; // context size
// computing the context size
ippsMD5GetSize(&ctxSize);
// context for the first half of the message
IppsMD5State* ctx1 = (IppsMD5State*)( new Ipp8u [ctxSize] );
// context  initialization
ippsMD5Init(ctx1);
// message example
Ipp8u msg[] = "Example of the MD5 message digest";
int m;
// updating the digest using the first half of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
 ippsMD5Update(msg+m, 1, ctx1);
// context for the entire message digest
```

```
IppsMD5State* ctx2=(IppsMD5State*)( new Ipp8u [ctxSize]);
ippsMD5Init(ctx2);
// context copying
ippsMD5Duplicate(ctx1, ctx2);
// digesting the first half of the message
Ipp8u digest[16];
ippsMD5Final(digest, ctx1);
// updating the digest
ippsMD5Update(msg+m, sizeof(msg)-1-m, ctx2);
// entire message digest
Ipp8u digest2[16];
ippsMD5Final(digest2, ctx2);
cout<<"MD5: ";
for(int k=0;k<16;k++){
 cout.fill('0');           // don't print n
 cout.width(2);            // instead of 0n
 cout<<hex<<(int)digest2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//MD5: a832466000c4488d94e6b3b508f176dd
```

6.4 SHA1 context preparation

To start SHA1 digest computations for streaming messages we have to prepare an appropriate context. The function `SHA1GetSize` shows the amount of memory we need.

```
IppStatus ippsSHA1GetSize(int* size)    //output:SHA1 context
                                             // size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `SHA1Init`.

```
IppStatus ippsSHA1Init(IppsSHA1State* ctx)//output:initialized
                                                  //SHA1 context
```

6.5 Using SHA1 primitives

The current input message can be digested with the use of `SHA1Update`.

```
 IppStatus ippsSHA1Update(const Ipp8u* msg,    //input:message
                        int msgLen,      //input:message length
                        IppsSHA1State* ctx)//input:SHA1 context
                                        //output:updated context
```

The `SHA1Duplicate` function duplicates the `IppsSHA1State` context.

```
IppStatus ippsSHA1Duplicate(IppsSHA1State* SrcCtx,    //input:
                                                //source context
                            IppsSHA1State* DstCtx)   //output:
                                             //destination context
```

The function `SHA1Final` completes calculation of the digest.

```
IppStatus ippsSHA1Final(Ipp8u* dgst,           //output:digest
                        IppsSHA1State* ctx) //input:SHA1 context
```

A call to the function `SHA1GetTag` retains the possibility to update the digest and computes the authentication tag.

```
IppStatus ippsSHA1GetTag(Ipp8u* tag,//output:authentication tag
                        IppsSHA1State* ctx) //input:SHA1 context
```

6.6 SHA1 example

```
// SHA1 digest of a streaming message:
// icpc -xHOST  -ipp=crypto  000Hash11.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
int ctxSize; // context size
```

```
// computing the context size
ippsSHA1GetSize(&ctxSize);
// context for the first half of the message
IppsSHA1State* ctx1=(IppsSHA1State*)( new Ipp8u [ctxSize]);
// context  initialization
ippsSHA1Init(ctx1);
// message example
Ipp8u msg[] = "Example of the SHA1 message digest";
int m;
// updating the digest using the first half of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
 ippsSHA1Update(msg+m, 1, ctx1);
// context for the entire message digest
IppsSHA1State* ctx2=(IppsSHA1State*)( new Ipp8u [ctxSize]);
ippsSHA1Init(ctx2);
// context copying
ippsSHA1Duplicate(ctx1, ctx2);
// digesting the first half of the message
Ipp8u digest[20];
ippsSHA1Final(digest, ctx1);
// updating the digest
ippsSHA1Update(msg+m, sizeof(msg)-1-m, ctx2);
// entire message digest
Ipp8u digest2[20];
ippsSHA1Final(digest2, ctx2);
cout<<"SHA1: ";
for(int k=0;k<20;k++){
 cout.fill('0');           // don't print n
 cout.width(2);            // instead of 0n
 cout<<hex<<(int)digest2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}
//Output:
//SHA1: af573fb2201b3be2094c38aaa3dc96b950dc473c
```

6.7 SHA224 context preparation

To start SHA224 digest computations for streaming messages we have to prepare an appropriate context. The function `SHA224GetSize` shows the amount of memory we need.

```
IppStatus ippsSHA224GetSize(int* size)//output:SHA224 context
                                              // size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `SHA224Init`.

```
IppStatus ippsSHA224Init(IppsSHA224State* ctx)      //output:
                                  //initialized SHA224 context
```

6.8 Using SHA224 primitives

The current input message can be digested with the use of `SHA224Update`.

```
IppStatus ippsSHA224Update(const Ipp8u* msg,   //input:message
                  int msgLen,          //input:message length
                  IppsSHA224State* ctx)//input:SHA224 context
                                    //output:updated context
```

The `SHA224Duplicate` function duplicates the `IppsSHA224State` context.

```
IppStatus ippsSHA224Duplicate(IppsSHA224State* SrcCtx,//input:
                                              //source context
                           IppsSHA224State* DstCtx) //output:
                                          //destination context
```

The function `SHA224Final` completes calculation of the digest.

```
IppStatus ippsSHA224Final(Ipp8u* dgst,         //output:digest
                  IppsSHA224State* ctx) //input:SHA224 context
```

A call to the function `SHA224GetTag` retains the possibility to update the digest and computes the authentication tag.

```
IppStatus ippsSHA224GetTag(Ipp8u* tag,              //output:
                                                 //authentication tag
                           IppsSHA224State* ctx) //input:SHA224
                                                 //context
```

6.9 SHA224 example

```
// SHA224 digest of a streaming message:
// icpc -xHOST  -ipp=crypto  000Hash12.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
int ctxSize; // context size
// computing the context size
ippsSHA224GetSize(&ctxSize);
// context for the first half of the message
IppsSHA224State* ctx1=(IppsSHA224State*)( new Ipp8u [ctxSize]);
// context  initialization
ippsSHA224Init(ctx1);
// message example
Ipp8u msg[] = "Example of the SHA224 message digest";
int m;
// updating the digest using the first half of the  message
for(m=0; m<(sizeof(msg)-1)/2; m++)
 ippsSHA224Update(msg+m, 1, ctx1);
// context for the entire message digest
IppsSHA224State* ctx2=(IppsSHA224State*)( new Ipp8u [ctxSize]);
ippsSHA224Init(ctx2);
// context copying
ippsSHA224Duplicate(ctx1, ctx2);
// digesting the first half of the message
Ipp8u digest[28];
ippsSHA224Final(digest, ctx1);
// updating the digest
ippsSHA224Update(msg+m, sizeof(msg)-1-m, ctx2);
// entire message digest
Ipp8u digest2[28];
```

```
ippsSHA224Final(digest2, ctx2);
cout<<"SHA224: ";
for(int k=0;k<28;k++){
 cout.fill('0');           // don't print n
 cout.width(2);            // instead of 0n
 cout<<hex<<(int)digest2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}
//Output:
//SHA224:
//5180baa766d38d7d501e73dd8895a4b20824d8ef625d81ff110ca43a
```

6.10 SHA256 context preparation

To start SHA256 digest computations for streaming messages we have to prepare an appropriate context. The function `SHA256GetSize` shows the amount of memory we need.

```
IppStatus ippsSHA256GetSize(int* size)//output:SHA256 context
                                              // size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `SHA256Init`.

```
IppStatus ippsSHA256Init(IppsSHA256State* ctx)      //output:
                                  //initialized SHA256 context
```

6.11 Using SHA256 primitives

The current input message can be digested with the use of `SHA256Update`.

```
IppStatus ippsSHA256Update(const Ipp8u* msg,   //input:message
                  int msgLen,          //input:message length
                  IppsSHA256State* ctx)//input:SHA256 context
                                       //output:updated context
```

The `SHA256Duplicate` function duplicates the `IppsSHA256State` context.

```
IppStatus ippsSHA256Duplicate(IppsSHA256State* SrcCtx,//input:
                                            //source context
                            IppsSHA256State* DstCtx) //output:
                                            //destination context
```

The function `SHA256Final` completes calculation of the digest.

```
IppStatus ippsSHA256Final(Ipp8u* dgst,         //output:digest
                  IppsSHA256State* ctx) //input:SHA256 context
```

A call to the function `SHA256GetTag` retains the possibility to update the digest and computes the authentication tag.

```
IppStatus ippsSHA256GetTag(Ipp8u* tag,              //output:
                                          //authentication tag
                         IppsSHA256State* ctx)  //input:SHA256
                                                //context
```

6.12 SHA256 example

```
// SHA256 digest of a streaming message:
// icpc -xHOST  -ipp=crypto  000Hash13.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
int ctxSize; // context size
// computing the context size
ippsSHA256GetSize(&ctxSize);
// context for the first half of the message
IppsSHA256State* ctx1=(IppsSHA256State*)( new Ipp8u [ctxSize]);
// context  initialization
```

```
ippsSHA256Init(ctx1);
// message example
Ipp8u msg[] = "Example of the SHA256 message digest";
int m;
// updating the digest using the first half of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
 ippsSHA256Update(msg+m, 1, ctx1);
// context for the entire message digest
IppsSHA256State* ctx2=(IppsSHA256State*)( new Ipp8u [ctxSize]);
ippsSHA256Init(ctx2);
// context copying
ippsSHA256Duplicate(ctx1, ctx2);
// digesting the first half of the message
Ipp8u digest[32];
ippsSHA256Final(digest, ctx1);
// updating the digest
ippsSHA256Update(msg+m, sizeof(msg)-1-m, ctx2);
// entire message digest
Ipp8u digest2[32];
ippsSHA256Final(digest2, ctx2);
cout<<"SHA256: ";
for(int k=0;k<32;k++){
 cout.fill('0');           // don't print n
 cout.width(2);            // instead of 0n
 cout<<hex<<(int)digest2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//SHA256: 0944039748e5093bce53295859e6f79621717b23ce
//        18b8f7d35d1c89781bb4d1
```

6.13 SHA384 context preparation

To start SHA384 digest computations for streaming messages we have to prepare an appropriate context. The function `SHA384GetSize` shows the amount of memory we need.

```
IppStatus ippsSHA384GetSize(int* size)//output:SHA384 context
                                              //size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `SHA384Init`.

```
IppStatus ippsSHA384Init(IppsSHA384State* ctx)      //output:
                                  //initialized SHA384 context
```

6.14 Using SHA384 primitives

The current input message can be digested with the use of `SHA384Update`.

```
IppStatus ippsSHA384Update(const Ipp8u* msg,    //input:message
                    int msgLen,          //input:message length
                    IppsSHA384State* ctx)//input:SHA384 context
                                        //output:updated context
```

The `SHA384Duplicate` function duplicates the `IppsSHA384State` context.

```
IppStatus ippsSHA384Duplicate(IppsSHA384State* SrcCtx,//input:
                                                //source context
                            IppsSHA384State* DstCtx) //output:
                                              //destination context
```

The function `SHA384Final` completes calculation of the digest.

```
IppStatus ippsSHA384Final(Ipp8u* dgst,         //output:digest
                    IppsSHA384State* ctx) //input:SHA384 context
```

A call to the function `SHA384GetTag` retains the possibility to update the digest and computes the authentication tag.

```
IppStatus ippsSHA384GetTag(Ipp8u* tag,                //output:
                                                  //authentication tag
                           IppsSHA384State* ctx)  //input:SHA384
                                                  //context
```

6.15 SHA384 example

```
// SHA384 digest of a streaming message:
// icpc -xHOST  -ipp=crypto  000Hash14.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
int ctxSize; // context size
// computing the context size
ippsSHA384GetSize(&ctxSize);
// context for the first half of message
IppsSHA384State* ctx1=(IppsSHA384State*)( new Ipp8u [ctxSize]);
// context  initialization
ippsSHA384Init(ctx1);
// message example
Ipp8u msg[] = "Example of the SHA384 message digest";
int m;
// updating the digest using the first half of message
for(m=0; m<(sizeof(msg)-1)/2; m++)
 ippsSHA384Update(msg+m, 1, ctx1);
// context for the entire message digest
IppsSHA384State* ctx2=(IppsSHA384State*)( new Ipp8u [ctxSize]);
ippsSHA384Init(ctx2);
// context copying
ippsSHA384Duplicate(ctx1, ctx2);
// digesting the first half of the message
Ipp8u digest[48];
ippsSHA384Final(digest, ctx1);
// updating the digest
ippsSHA384Update(msg+m, sizeof(msg)-1-m, ctx2);
// entire message digest
Ipp8u digest2[48];
```

```
ippsSHA384Final(digest2, ctx2);
cout<<"SHA384: ";
for(int k=0;k<48;k++){
 cout.fill('0');           // don't print n
 cout.width(2);            // instead of 0n
 cout<<hex<<(int)digest2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//SHA384: 7d68d966daa542bbdeb4d33abdfbba6973d263bc5eb
//e584a7747a03acf95a2de2cc6c1f6ea872ede60cccaeea9ba9662
```

6.16 SHA512 context preparation

To start SHA512 digest computations for streaming messages we have to prepare an appropriate context. The function `SHA512GetSize` shows the amount of memory we need.

```
IppStatus ippsSHA512GetSize(int* size)//output:SHA512 context
                                              // size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `SHA512Init`.

```
IppStatus ippsSHA512Init(IppsSHA512State* ctx)      //output:
                                  //initialized SHA512 context
```

6.17 Using SHA512 primitives

The current input message can be digested with the use of `SHA512Update`.

```
IppStatus ippsSHA512Update(const Ipp8u* msg,    //input:message
                    int msgLen,          //input:message length
                    IppsSHA512State* ctx)//input:SHA512 context
                                         //output:updated context
```

The `SHA512Duplicate` function duplicates the `IppsSHA512State` context.

```
IppStatus ippsSHA512Duplicate(IppsSHA512State* SrcCtx,//input:
                                              //source context
                              IppsSHA512State* DstCtx) //output:
                                              //destination context
```

The function `SHA512Final` completes the calculation of the digest.

```
IppStatus ippsSHA512Final(Ipp8u* dgst,         //output:digest
                    IppsSHA512State* ctx) //input:SHA512 context
```

A call to the function `SHA512GetTag` retains the possibility to update the digest and computes the authentication tag.

```
IppStatus ippsSHA512GetTag(Ipp8u* tag,               //output:
                                            //authentication tag
                           IppsSHA512State* ctx)  //input:SHA512
                                                  //context
```

6.18 SHA512 example

```
// SHA512 digest of a streaming message:
// icpc -xHOST  -ipp=crypto  000Hash15.cpp
#include "ippcp.h"
#include<iostream>
using namespace std;
int main(){
int ctxSize; // context size
// computing the context size
ippsSHA512GetSize(&ctxSize);
// context for the first half of message
IppsSHA512State* ctx1=(IppsSHA512State*)(new Ipp8u [ctxSize]);
// context  initialization
```

```
ippsSHA512Init(ctx1);
// message example
Ipp8u msg[] = "Example of the SHA512 message digest";
int m;
// updating the digest using the first half of message
for(m=0; m<(sizeof(msg)-1)/2; m++)
 ippsSHA512Update(msg+m, 1, ctx1);
// context for the entire message digest
IppsSHA512State* ctx2=(IppsSHA512State*)(new Ipp8u [ctxSize]);
ippsSHA512Init(ctx2);
// context copying
ippsSHA512Duplicate(ctx1, ctx2);
// digesting the first half of the message
Ipp8u digest[64];
ippsSHA512Final(digest, ctx1);
// updating the digest
ippsSHA512Update(msg+m, sizeof(msg)-1-m, ctx2);
// entire message digest
Ipp8u digest2[64];
ippsSHA512Final(digest2, ctx2);
cout<<"SHA512: ";
for(int k=0;k<64;k++){
 cout.fill('0');           // don't print n
 cout.width(2);            // instead of 0n
 cout<<hex<<(int)digest2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//SHA512:
//d6add6601f56785798da21e34d8996bd55a059847e168363b5428
//8a67e8de7d8771a4e86ce02e2fd6331a7214f578ae986300cfebb
//d18a9ee64aadaa46393425
```

Chapter 7

Message authentication functions for non-streaming messages

If the hashed message requires authentication, one can use HMAC (hashed message authentication code) mechanism. A detailed description of the algorithm can be found in [FIPS 198], [RFC 2202]. In the present chapter we restrict our considerations to the case of relatively short messages. In that case the entire message can be hashed at once.

7.1 HMACMD5MessageDigest

The `HMACMD5MessageDigest` function computes the MD5-based MAC value of an entire message. It takes the input key of a specified length and applies the keyed MD5-based MAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsHMACMD5MessageDigest(const Ipp8u* msg,  //input:
                                                      //message
                                int msgLen, //input:message length
                                const Ipp8u* key,      //input:key
                                Ipp8u* mac,           //output:MAC
                                int macLen)     //input:MAC length
```

```
// HMACMD5MessageDigest, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC00.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
```

```
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// message
Ipp8u msg[] = "what do ya want for nothing?";
int macLen = 16;
Ipp8u mac[16];
// HMACMD5
ippsHMACMD5MessageDigest(msg, strlen((char*)msg),
key, strlen((char*)key),mac, macLen);
cout<<"HMACMD5: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
}

//Output:
//HMACMD5: 750c783e6ab0b503eaa86e310a5db738
```

7.2 HMACSHA1MessageDigest

The `HMACSHA1MessageDigest` function computes the SHA1-based MAC value of an entire message. It takes the input key of a specified length and applies the keyed SHA1-based MAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsHMACSHA1MessageDigest(const Ipp8u* msg, //input:
                                                      //message
                          int msgLen,    //input:message length
                          const Ipp8u* key,         //input:key
                          Ipp8u* mac,              //output:MAC
                          int macLen)        //input:MAC length
```

```
// HMACSHA1MessageDigest, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC01.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// message
Ipp8u msg[] = "what do ya want for nothing?";
int macLen = 20;
Ipp8u mac[20];
// HMACSHA1
ippsHMACSHA1MessageDigest(msg, strlen((char*)msg),
key, strlen((char*)key),mac, macLen);
cout<<"HMACSHA1: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
}

//Output:
//HMACSHA1: effcdf6ae5eb2fa2d27416d5f184df9c259a7c79
```

7.3 HMACSHA224MessageDigest

The `HMACSHA224MessageDigest` function computes the SHA224-based MAC value of an entire message. It takes the input key of a specified length and applies the keyed SHA224-based MAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsHMACSHA224MessageDigest(const Ipp8u* msg,//input:
                                                        //message
                           int msgLen,    //input:message length
                           const Ipp8u* key,         //input:key
                           Ipp8u* mac,              //output:MAC
                           int macLen)        //input:MAC length
```

```
// HMACSHA224MessageDigest, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC02.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// message
Ipp8u msg[] = "what do ya want for nothing?";
int macLen = 28;
Ipp8u mac[28];
// HMACSHA224
ippsHMACSHA224MessageDigest(msg, strlen((char*)msg),
key, strlen((char*)key),mac, macLen);
cout<<"HMACSHA224: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
}

//Output:
//HMACSHA224:
//a30e01098bc6dbbf45690f3a7e9e6d0f8bbea2a39e6148008fd05e44
```

7.4 HMACSHA256MessageDigest

The `HMACSHA256MessageDigest` function computes the SHA256-based MAC value of an entire message. It takes the input key of a specified length and applies the keyed SHA256-based MAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsHMACSHA256MessageDigest(const Ipp8u* msg,//input:
                                                        //message
                           int msgLen,    //input:message length
                           const Ipp8u* key,         //input:key
                           Ipp8u* mac,              //output:MAC
                           int macLen)        //input:MAC length
```

```
// HMACSHA256MessageDigest, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC03.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// message
Ipp8u msg[] = "what do ya want for nothing?";
int macLen = 32;
Ipp8u mac[32];
// HMACSHA256
ippsHMACSHA256MessageDigest(msg, strlen((char*)msg),
key, strlen((char*)key),mac, macLen);
cout<<"HMACSHA256: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
```

```
}

//Output:
//HMACSHA256: 5bdcc146bf60754e6a042426089575c75a003f089d2739839d
//            ec58b964ec3843
```

7.5 HMACSHA384MessageDigest

The `HMACSHA384MessageDigest` function computes the SHA384-based MAC value of an entire message. It takes the input key of a specified length and applies the keyed SHA384-based MAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsHMACSHA384MessageDigest(const Ipp8u* msg,//input:
                                                      //message
                           int msgLen,    //input:message length
                           const Ipp8u* key,         //input:key
                           Ipp8u* mac,              //output:MAC
                           int macLen)        //input:MAC length
```

```
// HMACSHA384MessageDigest, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC04.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// message
Ipp8u msg[] = "what do ya want for nothing?";
int macLen = 48;
Ipp8u mac[48];
// HMACSHA384
ippsHMACSHA384MessageDigest(msg, strlen((char*)msg),
key, strlen((char*)key),mac, macLen);
cout<<"HMACSHA384: ";
for(int k=0;k<macLen;k++){
```

```
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
}

//Output:
//HMACSHA384:
//af45d2e376484031617f78d2b58a6b1b9c7ef464f5a01b47e42ec373632244
//5e8e2240ca5e69e2c78b3239ecfab21649
```

7.6 HMACSHA512MessageDigest

The `HMACSHA512MessageDigest` function computes the SHA512-based MAC value of an entire message. It takes the input key of a specified length and applies the keyed SHA512-based MAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsHMACSHA512MessageDigest(const Ipp8u* msg,//input:
                                                        //message
                           int msgLen,    //input:message length
                           const Ipp8u* key,         //input:key
                           Ipp8u* mac,              //output:MAC
                           int macLen)        //input:MAC length
```

```
// HMACSHA512MessageDigest, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC05.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// message
Ipp8u msg[] = "what do ya want for nothing?";
```

```
int macLen = 64;
Ipp8u mac[64];
// HMACSHA512
ippsHMACSHA512MessageDigest(msg, strlen((char*)msg),
key, strlen((char*)key),mac, macLen);
cout<<"HMACSHA512: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
}

//Output:
//HMACSHA512:
//164b7a7bfcf819e2e395fbe73b56e0a387bd64222e831fd610270cd7ea
//2505549758bf75c05a994a6d034f65f8f0e6fdcaeab1a34d4a6b4b636e
//070a38bce737
```

7.7 CMACRijndael128MessageDigest

The `CMACRijndael128MessageDigest` function is an example of a cipher-based message authentication code (CMAC). Its description can be found in [RFC 4493]. It computes the MAC value of an entire message using Rijndael128 block cipher. It takes the input key of a specified length and applies the keyed CMAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsCMACRijndael128MessageDigest(const Ipp8u* msg,
                                                  //input:message
                          int msgLen,    //input:message length
                          const Ipp8u* key,         //input:key
                          IppsRijndaelKeyLength keyLen,//input:
                                                  //key length
                          Ipp8u* mac,               //output:MAC
                          int macLen)        //input:MAC length
```

```
// CMACRijndael128MessageDigest, rfc4493 example
// icpc -xHOST  -ipp=crypto  000CMAC00.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){

// key
Ipp8u key[]= {0x2b,0x7e,0x15,0x16,0x28,0xae,0xd2,0xa6,
              0xab,0xf7,0x15,0x88,0x09,0xcf,0x4f,0x3c};
// message
Ipp8u msg[] = "";
int macLen = 16;
Ipp8u mac[16];
// CMACRijndael128
ippsCMACRijndael128MessageDigest(msg, strlen((char*)msg),
key,(IppsRijndaelKeyLength)128 ,mac, macLen);
cout<<"CMACRijndael128: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac[k];
}
cout<<endl;
return 0;
}

//Output:
//CMACRijndael128: bb1d6929e95937287fa37d129b756746
```

7.8 XCBCRijndael128MessageTag

The `XCBCRijndael128MessageTag` function is another example of a cipher-based message authentication code (CMAC). It is described in [RFC 3566]. It contains a set of extensions to the classical CMAC algorithm which overcome some limitations related to messages of varying lengths. It computes

the MAC value of an entire message using Rijndael128 block cipher. It takes the input key of a specified length and applies the keyed CMAC algorithm to the message. As a result we obtain the message authentication code (MAC) of the specified length.

```
IppStatus ippsXCBCRijndael128MessageTag(const Ipp8u* msg,
                                                  //input:message
                          Ipp32u msgLen, //input:message length
                          const Ipp8u* key,         //input:key
                          IppsRijndaelKeyLength keyLen,//input:
                                                  //key length
                          Ipp8u* tag,              //output:tag
                          int tagLen)        //input:tag length
```

```
// XCBCRijndael128MessageDigest, rfc3566 example
// icpc -xHOST  -ipp=crypto  000XCBC00.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[]= {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
// message
Ipp8u msg[] = "";
Ipp32u tagLen = 16;
Ipp8u tag[16];
// XCBCRijndael128
ippsXCBCRijndael128MessageTag(msg, (Ipp32u)strlen((char*)msg),
key,(IppsRijndaelKeyLength)128 ,tag, tagLen);
cout<<"XCBCRijndael128: ";
for(int k=0;k<tagLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)tag[k];
}
cout<<endl;
```

```
return 0;
}
//Output:
//XCBCRijndael128: 75f0251d528ac01c4573dfd584d79f29
```

Chapter 8

Message authentication functions for streaming messages

8.1 HMACMD5 context preparation

To start HMACMD5 computations for streaming messages we have to prepare an appropriate context. The function `HMACMD5GetSize` shows the amount of memory we need.

```
IppStatus ippsHMACMD5GetSize(int* size)     //output:HMACMD5
                                          //context size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `HMACMD5Init`.

```
IppStatus ippsHMACMD5Init(const Ipp8u key,         //input:key
                          int keyLen,      //input:key length
                          IppsHMACMD5State* ctx)    //output:
                                //initialized HMACMD5 context
```

8.2 Using HMACMD5 primitives

The current input message can be digested with the use of `HMACMD5Update`.

```
IppStatus ippsHMACMD5Update(const Ipp8u* msg, //input:message
                int msgLen,            //input:message length
                IppsHMACMD5State* ctx)//input:HMACMD5 context
                                      //output:updated context
```

The function `HMACMD5Duplicate` duplicates the `IppsHMACMD5State` context.

```
IppStatus ippsHMACMD5Duplicate(IppsHMACMD5State* ctx1,//input:
                                                 //source context
                               IppsHMACMD5State* ctx2) //output:
                                                 //destination context
```

The function `HMACMD5Final` completes calculation of the MAC.

```
IppStatus ippsHMACMD5Final(Ipp8u* MAC,             //output:MAC
                           IppsHMACMD5State* ctx) //input:HMACMD5
                                                  //context
```

A call to the function `HMACMD5GetTag` retains the possibility to update the MAC and computes the authentication tag.

```
IppStatus ippsHMACMD5GetTag(Ipp8u* tag,               //output:
                                                //authentication tag
                            IppsHMACMD5State* ctx) //input:HMACMD5
                                                  //context
```

8.3 HMACMD5 example

```
// HMACMD5 for a streaming message, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC10.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// size of HMACMD5 context
int ctxSize;
ippsHMACMD5GetSize(&ctxSize);
IppsHMACMD5State* ctx1=(IppsHMACMD5State*)(new Ipp8u [ctxSize]);
// first context initialization
ippsHMACMD5Init(key, strlen((char*)key), ctx1);
```

```
// message
Ipp8u msg[] = "what do ya want for nothing?";
int m;
// updating MAC using the first part of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
ippsHMACMD5Update(msg+m, 1, ctx1);
// duplication
IppsHMACMD5State* ctx2=(IppsHMACMD5State*)(new Ipp8u [ctxSize]);
ippsHMACMD5Init(key, strlen((char*)key), ctx2);
ippsHMACMD5Duplicate(ctx1, ctx2);
// final MAC of the first part
const int macLen = 16;
Ipp8u mac[macLen];
ippsHMACMD5Final(mac, macLen, ctx1);
// updating MAC using the second context
ippsHMACMD5Update(msg+m, sizeof(msg)-1-m, ctx2);
// final MAC of the complete message
Ipp8u mac2[macLen];
ippsHMACMD5Final(mac2, macLen, ctx2);
cout<<"HMACMD5: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//HMACMD5: 750c783e6ab0b503eaa86e310a5db738
```

8.4 HMACSHA1 context preparation

To start HMACSHA1 computations for streaming messages we have to prepare an appropriate context. The function `HMACSHA1GetSize` shows the

amount of memory we need.

```
IppStatus ippsHMACSHA1GetSize(int* size)   //output:HMACSHA1
                                          //context size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `HMACSHA1Init`.

```
IppStatus ippsHMACSHA1Init(const Ipp8u key,      //input:key
                           int keyLen,    //input:key length
                           IppsHMACSHA1State* ctx) //output:
                              //initialized HMACSHA1 context
```

8.5 Using HMACSHA1 primitives

The current input message can be digested with the use of `HMACSHA1Update`.

```
IppStatus ippsHMACSHA1Update(const Ipp8u* msg,//input:message
              int msgLen,              //input:message length
              IppsHMACSHA1State* ctx)//input:HMACSHA1 context
                                       //output:updated context
```

The function `HMACSHA1Duplicate` duplicates the `IppsHMACSHA1State` context.

```
IppStatus ippsHMACSHA1Duplicate(IppsHMACSHA1State* ctx1,//inp.:
                                          //source context
                            IppsHMACSHA1State* ctx2) //output:
                                        //destination context
```

The function `HMACSHA1Final` completes the calculation of the MAC.

```
IppStatus ippsHMACSHA1Final(Ipp8u* MAC,          //output:MAC
                       IppsHMACSHA1State* ctx)//input:HMACSHA1
                                                //context
```

A call to the function `HMACSHA1GetTag` retains the possibility to update the MAC and computes the authentication tag.

```
IppStatus ippsHMACSHA1GetTag(Ipp8u* tag,              //output:
                                          //authentication tag
                        IppsHMACSHA1State* ctx)//input:HMACSHA1
                                                  //context
```

8.6 HMACSHA1 example

```
// HMACSHA1 for a streaming message, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC11.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// size of HMACSHA1 context
int ctxSize;
ippsHMACSHA1GetSize(&ctxSize);
IppsHMACSHA1State* ctx1=(IppsHMACSHA1State*)(new Ipp8u [ctxSize]);
// first context initialization
ippsHMACSHA1Init(key, strlen((char*)key), ctx1);
// message
Ipp8u msg[] = "what do ya want for nothing?";
int m;
// updating MAC using the first part of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
ippsHMACSHA1Update(msg+m, 1, ctx1);
// duplication
IppsHMACSHA1State* ctx2=(IppsHMACSHA1State*)(new Ipp8u [ctxSize]);
ippsHMACSHA1Init(key, strlen((char*)key), ctx2);
ippsHMACSHA1Duplicate(ctx1, ctx2);
// final MAC of the first part
const int macLen = 20;
Ipp8u mac[macLen];
ippsHMACSHA1Final(mac, macLen, ctx1);
// updating MAC using the second context
ippsHMACSHA1Update(msg+m, sizeof(msg)-1-m, ctx2);
```

```
// final MAC of the complete message
Ipp8u mac2[macLen];
ippsHMACSHA1Final(mac2, macLen, ctx2);
cout<<"HMACSHA1: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//HMACSHA1: effcdf6ae5eb2fa2d27416d5f184df9c259a7c79
```

8.7 HMACSHA224 context preparation

To start HMACSHA224 computations for streaming messages we have to prepare an appropriate context. The function `HMACSHA224GetSize` shows the amount of memory we need.

```
IppStatus ippsHMACSHA224GetSize(int* size)//output:HMACSHA224
                                          //context size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `HMACSHA224Init`.

```
IppStatus ippsHMACSHA224Init(const Ipp8u key,     //input:key
                          int keyLen,      //input:key length
                          IppsHMACSHA224State* ctx)  //output:
                             //initialized HMACSHA224 context
```

8.8 Using HMACSHA224 primitives

The current input message can be digested with the use of `HMACSHA224Update`.

```
IppStatus ippsHMACSHA224Update(const Ipp8u* msg,//input:message
            int msgLen,                    //input:message length
            IppsHMACSHA224State* ctx) //input:HMACSHA224 context
                                      //output:updated context
```

The function `HMACSHA224Duplicate` duplicates the `IppsHMACSHA224State` context.

```
IppStatus ippsHMACSHA224Duplicate(IppsHMACSHA224State* ctx1,
                                        //input:source context
                            IppsHMACSHA224State* ctx2)//output:
                                        //destination context
```

The function `HMACSHA224Final` completes calculation of the MAC.

```
IppStatus ippsHMACSHA224Final(Ipp8u* MAC,          //output:MAC
                            IppsHMACSHA224State* ctx)  //input:
                                        //HMACSHA224 context
```

A call to the function `HMACSHA224GetTag` retains the possibility to update the MAC and computes the authentication tag.

```
IppStatus ippsHMACSHA224GetTag(Ipp8u* tag,           //output:
                                        //authentication tag
                            IppsHMACSHA224State* ctx)  //input:
                                        //HMACSHA224 context
```

8.9 HMACSHA224 example

```
// HMACSHA224 for a streaming message, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC12.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
```

```
int main(){
// key
Ipp8u key[] = "Jefe";
// size of HMACSHA224 context
int ctxSize;
ippsHMACSHA224GetSize(&ctxSize);
IppsHMACSHA224State* ctx1=(IppsHMACSHA224State*)
                                    (new Ipp8u [ctxSize]);
// first context initialization
ippsHMACSHA224Init(key, strlen((char*)key), ctx1);
// message
Ipp8u msg[] = "what do ya want for nothing?";
int m;
// updating MAC using the first part of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
ippsHMACSHA224Update(msg+m, 1, ctx1);
// duplication
IppsHMACSHA224State* ctx2=(IppsHMACSHA224State*)
                                      (new Ipp8u [ctxSize]);
ippsHMACSHA224Init(key, strlen((char*)key), ctx2);
ippsHMACSHA224Duplicate(ctx1, ctx2);
// final MAC of the first part
const int macLen = 28;
Ipp8u mac[macLen];
ippsHMACSHA224Final(mac, macLen, ctx1);
// updating MAC using the second context
ippsHMACSHA224Update(msg+m, sizeof(msg)-1-m, ctx2);
// final MAC of the complete message
Ipp8u mac2[macLen];
ippsHMACSHA224Final(mac2, macLen, ctx2);
cout<<"HMACSHA224: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
```

```
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//HMACSHA224:
//a30e01098bc6dbbf45690f3a7e9e6d0f8bbea2a39e6148008fd05e44
```

8.10 HMACSHA256 context preparation

To start HMACSHA256 computations for streaming messages we have to prepare an appropriate context. The function `HMACSHA256GetSize` shows the amount of memory we need.

```
IppStatus ippsHMACSHA256GetSize(int* size)//output:HMACSHA256
                                          //context size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `HMACSHA256Init`.

```
IppStatus ippsHMACSHA256Init(const Ipp8u key,     //input:key
                             int keyLen,     //input:key length
                             IppsHMACSHA256State* ctx) //output:
                               //initialized HMACSHA256 context
```

8.11 Using HMACSHA256 primitives

The current input message can be digested with the use of `HMACSHA256Update`.

```
IppStatus ippsHMACSHA256Update(const Ipp8u* msg,//input:message
            int msgLen,                   //input:message length
            IppsHMACSHA256State* ctx) //input:HMACSHA256 context
                                       //output:updated context
```

The function `HMACSHA256Duplicate` duplicates the `IppsHMACSHA256State` context.

```
IppStatus ippsHMACSHA256Duplicate(IppsHMACSHA256State* ctx1,
                                        //input:source context
                            IppsHMACSHA256State* ctx2)//output:
                                         //destination context
```

The function `HMACSHA256Final` completes calculation of the MAC.

```
IppStatus ippsHMACSHA256Final(Ipp8u* MAC,         //output:MAC
                            IppsHMACSHA256State* ctx)  //input:
                                          //HMACSHA256 context
```

A call to the function `HMACSHA256GetTag` retains the possibility to update the MAC and computes the authentication tag.

```
IppStatus ippsHMACSHA256GetTag(Ipp8u* tag,           //output:
                                        //authentication tag
                            IppsHMACSHA256State* ctx)  //input:
                                          //HMACSHA256 context
```

8.12 HMACSHA256 example

```
// HMACSHA256 for a streaming message, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC13.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// size of HMACSHA256 context
int ctxSize;
ippsHMACSHA256GetSize(&ctxSize);
IppsHMACSHA256State* ctx1 = (IppsHMACSHA256State*)
                                    ( new Ipp8u [ctxSize] );
// first context initialization
ippsHMACSHA256Init(key, strlen((char*)key), ctx1);
// message
```

```
Ipp8u msg[] = "what do ya want for nothing?";
int m;
// updating MAC using the first part of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
ippsHMACSHA256Update(msg+m, 1, ctx1);
// duplication
IppsHMACSHA256State* ctx2= (IppsHMACSHA256State*)
                                    ( new Ipp8u [ctxSize] );
ippsHMACSHA256Init(key, strlen((char*)key), ctx2);
ippsHMACSHA256Duplicate(ctx1, ctx2);
// final MAC of the first part
const int macLen = 32;
Ipp8u mac[macLen];
ippsHMACSHA256Final(mac, macLen, ctx1);
// updating MAC using the second context
ippsHMACSHA256Update(msg+m, sizeof(msg)-1-m, ctx2);
// final MAC of the complete message
Ipp8u mac2[macLen];
ippsHMACSHA256Final(mac2, macLen, ctx2);
cout<<"HMACSHA256: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//HMACSHA256: 5bdcc146bf60754e6a042426089575c75a003f089d273983
//            9dec58b964ec3843
```

8.13 HMACSHA384 context preparation

To start HMACSHA384 computations for streaming messages we have to prepare an appropriate context. The function `HMACSHA384GetSize` shows the amount of memory we need.

```
IppStatus ippsHMACSHA384GetSize(int* size)//output:HMACSHA384
                                          //context size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `HMACSHA384Init`.

```
IppStatus ippsHMACSHA384Init(const Ipp8u key,     //input:key
                          int keyLen,      //input:key length
                          IppsHMACSHA384State* ctx)  //output:
                             //initialized HMACSHA384 context
```

8.14 Using HMACSHA384 primitives

The current input message can be digested with the use of `HMACSHA384Update`.

```
IppStatus ippsHMACSHA384Update(const Ipp8u* msg,//input:message
           int msgLen,                   //input:message length
           IppsHMACSHA384State* ctx) //input:HMACSHA384 context
                                     //output:updated context
```

The function `HMACSHA384Duplicate` duplicates the `IppsHMACSHA384State` context.

```
IppStatus ippsHMACSHA384Duplicate(IppsHMACSHA384State* ctx1,
                                     //input:source context
                          IppsHMACSHA384State* ctx2)//output:
                                     //destination context
```

The function `HMACSHA384Final` completes calculation of the MAC.

```
IppStatus ippsHMACSHA384Final(Ipp8u* MAC,         //output:MAC
                          IppsHMACSHA384State* ctx)  //input:
                                       //HMACSHA384 context
```

A call to the function `HMACSHA384GetTag` retains the possibility to update the MAC and computes the authentication tag.

```
IppStatus ippsHMACSHA384GetTag(Ipp8u* tag,          //output:
                                          //authentication tag
                          IppsHMACSHA384State* ctx)  //input:
                                          //HMACSHA384 context
```

8.15 HMACSHA384 example

```
// HMACSHA384 for a streaming message, rfc2202 example
// icpc -xHOST  -ipp=crypto  000HMAC14.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// size of HMACSHA384 context
int ctxSize;
ippsHMACSHA384GetSize(&ctxSize);
IppsHMACSHA384State* ctx1 = (IppsHMACSHA384State*)
                                    ( new Ipp8u [ctxSize] );
// first context initialization
ippsHMACSHA384Init(key, strlen((char*)key), ctx1);
// message
Ipp8u msg[] = "what do ya want for nothing?";
int m;
// updating MAC using the first part of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
ippsHMACSHA384Update(msg+m, 1, ctx1);
// duplication
IppsHMACSHA384State* ctx2= (IppsHMACSHA384State*)
                                    ( new Ipp8u [ctxSize] );
ippsHMACSHA384Init(key, strlen((char*)key), ctx2);
ippsHMACSHA384Duplicate(ctx1, ctx2);
```

```
// final MAC of the first part
const int macLen = 48;
Ipp8u mac[macLen];
ippsHMACSHA384Final(mac, macLen, ctx1);
// updating MAC using the second context
ippsHMACSHA384Update(msg+m, sizeof(msg)-1-m, ctx2);
// final MAC of the complete message
Ipp8u mac2[macLen];
ippsHMACSHA384Final(mac2, macLen, ctx2);
cout<<"HMACSHA384: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)mac2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//HMACSHA384: af45d2e376484031617f78d2b58a6b1b9c7ef464f5a01b47e
//            42ec3736322445e8e2240ca5e69e2c78b3239ecfab21649
```

8.16 HMACSHA512 context preparation

To start HMACSHA512 computations we have to prepare an appropriate context. The function `HMACSHA512GetSize` shows the amount of memory we need.

```
IppStatus ippsHMACSHA512GetSize(int* size)//output:HMACSHA512
                                          //context size in bytes
```

The obtained value allows for an appropriate memory allocation. The context can be initialized with the use of `HMACSHA512Init`.

```
IppStatus ippsHMACSHA512Init(const Ipp8u key,     //input:key
                             int keyLen,     //input:key length
                             IppsHMACSHA512State* ctx)//output:
                               //initialized HMACSHA512 context
```

8.17 Using HMACSHA512 primitives

The current input message can be digested with the use of `HMACSHA512Update`.

```
IppStatus ippsHMACSHA512Update(const Ipp8u* msg,//input:message
          int msgLen,                    //input:message length
          IppsHMACSHA512State* ctx)  //input:HMACSHA512 context
                                      //output:updated context
```

The function `HMACSHA512Duplicate` duplicates the `IppsHMACSHA512State` context.

```
IppStatus ippsHMACSHA512Duplicate(IppsHMACSHA512State* ctx1,
                                        //input:source context
                            IppsHMACSHA512State* ctx2)//output:
                                        //destination context
```

The function `HMACSHA512Final` completes calculation of the MAC.

```
IppStatus ippsHMACSHA512Final(Ipp8u* MAC,         //output:MAC
                            IppsHMACSHA512State* ctx)  //input:
                                        //HMACSHA512 context
```

A call to the function `HMACSHA512GetTag` retains the possibility to update the MAC and computes the authentication tag.

```
IppStatus ippsHMACSHA512GetTag(Ipp8u* tag,           //output:
                                        //authentication tag
                            IppsHMACSHA512State* ctx)  //input:
                                        //HMACSHA512 context
```

8.18 HMACSHA512 example

```
// HMACSHA512 for a streaming message, rfc2202 example
```

```
// icpc -xHOST  -ipp=crypto  000HMAC15.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] = "Jefe";
// size of HMACSHA512 context
int ctxSize;
ippsHMACSHA512GetSize(&ctxSize);
IppsHMACSHA512State* ctx1 = (IppsHMACSHA512State*)
                                    ( new Ipp8u [ctxSize] );
// first context initialization
ippsHMACSHA512Init(key, strlen((char*)key), ctx1);
// message
Ipp8u msg[] = "what do ya want for nothing?";
int m;
// updating MAC using the first part of the message
for(m=0; m<(sizeof(msg)-1)/2; m++)
ippsHMACSHA512Update(msg+m, 1, ctx1);
// duplication
IppsHMACSHA512State* ctx2= (IppsHMACSHA512State*)
                                    ( new Ipp8u [ctxSize] );
ippsHMACSHA512Init(key, strlen((char*)key), ctx2);
ippsHMACSHA512Duplicate(ctx1, ctx2);
// final MAC of the first part
const int macLen = 64;
Ipp8u mac[macLen];
ippsHMACSHA512Final(mac, macLen, ctx1);
// updating MAC using the second context
ippsHMACSHA512Update(msg+m, sizeof(msg)-1-m, ctx2);
// final MAC of the complete message
Ipp8u mac2[macLen];
ippsHMACSHA512Final(mac2, macLen, ctx2);
cout<<"HMACSHA512: ";
for(int k=0;k<macLen;k++){
cout.fill('0');
```

```
cout.width(2);
cout<<hex<<(int)mac2[k];
}
cout<<endl;
delete [] (Ipp8u*)ctx1;
delete [] (Ipp8u*)ctx2;
return 0;
}

//Output:
//HMACSHA512:
//164b7a7bfcf819e2e395fbe73b56e0a387bd64222e831fd610270cd7ea2
//505549758bf75c05a994a6d034f65f8f0e6fdcaeab1a34d4a6b4b636e07
//0a38bce737
```

Chapter 9

Data Authentication Functions

The Data Authentication Algorithm (DAA) can be used as a tool against both accidental and intentional data modification. The algorithm is described in [FIPS 113]. All functions from this chapter use an input key of a specified length and specified block cipher to compute a data authentication code (DAC) of a specified length.

9.1 DAARijndael128MessageDigest

The function `DAARijndael128MessageDigest` applies the Rijndael128 algorithm with a specified key to compute the DAC of a specified length.

```
IppStatus ippsDAARijndael128MessageDigest(const Ipp8u* msg,
                                                  //input:message
                          int msgLen,    //input:message length
                          const Ipp8u* key,         //input:key
                          IppsRijndaelKeyLength keyLen,//input:
                                                  // key length
                          Ipp8u* dac,              //output:DAC
                          int dacLen)        //input:DAC length
```

```
// DAARijndael128MessageDigest
// icpc -xHOST  -ipp=crypto  000DAA02.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
```

```
int main(){
// key
Ipp8u key[] =
{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
0x05,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
// message
Ipp8u msg[] = "7654321 Now is the time for ";
int dacLen = 16;
Ipp8u dac[16];
// DAARijndael128 DAC
ippsDAARijndael128MessageDigest(msg, strlen((char*)msg),
key,(IppsRijndaelKeyLength)128,dac, dacLen);
cout<<"DAARijndael128 DAC: ";
for(int k=0;k<dacLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)dac[k];
}
cout<<endl;
return 0;
}
//Output:
//DAARijndael128 DAC: 8964f469c0abd22ce861fd6cf264e427
```

9.2 DAARijndael192MessageDigest

The function `DAARijndael192MessageDigest` applies the Rijndael192 algorithm with a specified key to compute the DAC of a specified length.

```
IppStatus ippsDAARijndael192MessageDigest(const Ipp8u* msg,
                                          //input:message
                          int msgLen,    //input:message length
                          const Ipp8u* key,         //input:key
                          IppsRijndaelKeyLength keyLen,//input:
                                            // key length
                          Ipp8u* dac,              //output:DAC
                          int dacLen)        //input:DAC length
```

```
// DAARijndael192MessageDigest
// icpc -xHOST  -ipp=crypto  000DAA03.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] =
{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
0x02,0x01,0x03,0x04,0x05,0x06,0x07,0x08,
0x05,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
// message
Ipp8u msg[] = "7654321 Now is the time for ";
int dacLen = 24;
Ipp8u dac[24];
// DAARijndael192 DAC
ippsDAARijndael192MessageDigest(msg, strlen((char*)msg),
key,(IppsRijndaelKeyLength)192,dac, dacLen);
cout<<"DAARijndael192 DAC: ";
for(int k=0;k<dacLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)dac[k];
}
cout<<endl;
return 0;
}

//Output:
//DAARijndael192 DAC:
//ec8140df9eb308091b925989f9dc605ecc7faad8ce159fbf
```

9.3 DAARijndael256MessageDigest

The function `DAARijndael256MessageDigest` applies the Rijndael256 algorithm with a specified key to compute the DAC of a specified length.

```
IppStatus ippsDAARijndael256MessageDigest(const Ipp8u* msg,
                                                  //input:message
                          int msgLen,    //input:message length
                          const Ipp8u* key,         //input:key
                          IppsRijndaelKeyLength keyLen,//input:
                                                  // key length
                          Ipp8u* dac,              //output:DAC
                          int dacLen)        //input:DAC length
```

```
// DAARijndael256MessageDigest
// icpc -xHOST  -ipp=crypto  000DAA04.cpp
#include "ippcp.h"
#include<iostream>
#include<cstring>
using namespace std;
int main(){
// key
Ipp8u key[] =
{0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
0x02,0x01,0x03,0x04,0x05,0x06,0x07,0x08,
0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
0x05,0x02,0x03,0x04,0x05,0x06,0x07,0x08};
// message
Ipp8u msg[] = "7654321 Now is the time for ";
int dacLen = 32;
Ipp8u dac[32];
// DAARijndael256 DAC
ippsDAARijndael256MessageDigest(msg, strlen((char*)msg),
key,(IppsRijndaelKeyLength)256,dac, dacLen);
cout<<"DAARijndael256 DAC: ";
for(int k=0;k<dacLen;k++){
cout.fill('0');
cout.width(2);
cout<<hex<<(int)dac[k];
}
cout<<endl;
return 0;
}
```

```
//Output:
//DAARijndael256 DAC:
//ae01991b5db937ac4ee06b7d86968a777c886143b9d1ad34fd6279b659
//7cdc45
```

Part II

Public key cryptography

Chapter 10

Big Numbers in IPP

10.1 The notion of `BigNum`

A general description of big integers arithmetics is given in [HAC], chapter 14.
The IPP library contains an efficient implementation of big integers arithmetic with both signed and unsigned versions. In the sequel we restrict our attention to the signed big integers (`BigNum`).

The `BigNum` in IPP is given by the sign and an array of 32-bit unsigned integers i.e integers of the type `Ipp32u`. The magnitude of a Big number is equal to

$$\sum_{i=0}^{N-1} a_i * 2^{32*i}, \quad \{a_0, \ldots, a_{N-1}\} - \text{ an array of unsigned 32-bit integers.}$$

Using the hexadecimal representation of the 32-bit components we can write 128-bit `BigNum` example as

{`0xabcdef12, 0x87654321, 0x12345678, 0x21fedcba`}.

Note that the least significant 32-bit word goes first.
Sometimes the octet string representation of the `BigNum` is preferred. The `BigNum` from the last example has the following octet string representation:

```
"\x21\xfe\xdc\xba\x12\x34\x56\x78\x87\x65\x43\x21\xab\xcd\xef\x12"
```

Note that this time the most significant byte goes first.

10.2 The `BigNum` setup

To set a `BigNum` up we have to perform the following steps.
1. Compute the size of `BigNum` context using `BigNumGetSize`.

```
IppStatus ippsBigNumGetSize (int dS,     //input:  data Size
                                           //in Ipp32u words
                             int* cS)  //output:  BN context
                                         //size in bytes
```

2. Allocate the memory for `BigNum` context using obtained `BigNum` size `cS`.

3. Initialize the `BigNum` using `BigNumInit`.

```
IppStatus ippsBigNumInit (int dS,                //input:data
                                        //size in Ipp32u words
                          IppsBigNumState* BN)  //initialized
                                                 //BN-context
```

4. Set the sign and the value of `BigNum` using `Set_BN`.

```
IppStatus ippsSet_BN (IppsBigNumSGN sgn,          //input:sign
                      int dS,                 //input:data Size
                      const Ipp32u* A,       //input:data array
                      IppsBigNumState* BN)    //out:BN-context
```

All the steps described above are performed by the function `nBN`.

```
IppsBigNumState* nBN (int dS,            //input:data Size in
                                               //Ipp32u words
                      const Ipp32u* A)  //input:data array of
                                               //Ipp32u words
```

Remark. `nBN` is not a part of IPP. It is defined to simplify the examples.

```
// function creating a new BigNum
IppsBigNumState* nBN(int dS, const Ipp32u* A)
{                    // dS -data Size in Ipp32u words,
   int cS;           // A -data Array, cS -BigNum context Size
   ippsBigNumGetSize(dS, &cS);  // get  BigNum context Size
```

```
   IppsBigNumState* BN = (IppsBigNumState*)( new Ipp8u [cS] );
   ippsBigNumInit(dS, BN);   // BigNum context initialization
   if(A)
      ippsSet_BN(IppsBigNumPOS, dS, A, BN);// set BigNum sign
   return BN;                                // and value
}
```

Let us present our first working example.

```
// 014Set_BN.cpp, Example of setting BigNum
// icpc -ipp=crypto 014Set_BN.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h"    //  nBN,tBN  fun. declarations

int main(){
using namespace std ;
Ipp32u A[] = {0xabcdef12, 0x87654321, 0x12345678, 0x21fedcba};
int dS=(sizeof A)/(sizeof(Ipp32u));// data Size in Ipp32u words
IppsBigNumState* BNA=nBN(dS,A);    // creating a new BigNum
tBN("BNA: ",BNA);                  // Typing BigNum
delete [] (Ipp8u*) BNA;            // freeing memory
return 0;}

// Output:
// BNA: 21fedcba1234567887654321abcdef12
```

Remarks.
1. We have used also the typing function `tBN`, which will be defined later. All support functions declarations and definitions can be found in the files `toolb.h, toolb.cpp` included in the Appendix.
2. The sign of `BigNum` can take the values `IppsBigNumPOS` or `IppsBigNumNEG`.

One can also define a `BigNum` using octet string representation (a sequence of bytes).

```
IppStatus ippsSetOctString_BN  (const Ipp8u* A,//input:oct-str
                                int dA,          //input:A-Size
                                IppsBigNumState* BN)  //BN-cont
```

```
// Setting up a BigNum using an octet string representation
// (sequence of 8bit elements)
// icpc -ipp=crypto 016SetOctString_BN.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
using namespace std ;

Ipp8u A[] = "\x21\xfe\xdc\xba\x12\x34\x56\x78"
                                "\x87\x65\x43\x21\xab\xcd\xef\x12";
int size = (sizeof(A)-1+3)/(sizeof(Ipp32u));
IppsBigNumState* BNA = nBN(size);
ippsSetOctString_BN(A, sizeof(A)-1, BNA);
tBN("BNA:\n", BNA);
delete [] (Ipp8u*)BNA;
return 0;}

// Output:
// BNA:
// 21fedcba1234567887654321abcdef12
```

10.3 Getting information from BigNum context

When the BigNum is defined one can obtain its characteristics using one of the functions

1. GetSize_BN

```
IppStatus ippsGetSize_BN  (const IppsBigNumState* BN,//input:
                                                    //BN-context
                                int* sBN)    //output:BN-size
```

2. Get_BN

```
IppStatus ippsGet_BN  (IppsBigNumSGN* sgn,     //outp:BN-sign
                  int* sBN,                    //outp:BN-size
                  Ipp32u* dBN,                 //outp:BN-data
                  const IppsBigNumState* BN) //inp:BN-context
```

3. ExtGet_BN

```
IppStatus ippsExtGet_BN(IppsBigNumSGN* sgn,   //output:BN-sign
                  int* sBits,             //output:BN bit-size
                  Ipp32u* dBN,                //output:BN-data
                  const IppsBigNumState* BN) //input:BN-context
```

4. Ref_BN

```
IppStatus ippsRef_BN (IppsBigNumSGN* sgn,     //output:BN-sign
                  int* sBits,             //output:BN bit-size
                  Ipp32u** dBN,               //output:BN-data
                  const IppsBigNumState* BN) //input/BN-context
```

5. GetOctStr_BN

```
IppStatus ippsGetOctString_BN(const Ipp8u* vBN,      //output:
                                                  //BN byte-array
                  int* sBytes,           //output:BN byte-size
                  const IppsBigNumState* BN) //input:BN-context
```

Here is an example how to use the first three functions.

```
// Getting BigNum sign, value, size
// icpc -ipp=crypto 017Get_BN.cpp toolb.cpp
#include<iostream>
#include "ippcp.h"
#include "toolb.h"    //  nBN, tBN  fun. declaration

int main(){
using namespace std ;
Ipp32u A[] = {0x11234567,0x1abcdeff,0x11234567,
                         0x1abcdeff,0xf1223344};
```

```
int dS=(sizeof A)/(sizeof(Ipp32u));// data Size in Ipp32u words
IppsBigNumState* BNA=nBN(dS,A);    // creating new BigNum
int sBN;      // Big Number size in Ipp32u-words
ippsGetSize_BN(BNA,&sBN);          // get BigNum size
cout << "BNA size: "<< sBN << endl;
IppsBigNumSGN sgn;
Ipp32u* dBN=new Ipp32u[sBN];   // array of Ipp32components

ippsGet_BN(&sgn,&sBN,dBN,BNA); // get BigNum sign and value
cout << "From Get_BN:" << endl;
cout<<"BNA data: "<< hex;
for(int k=0;k<sBN;k++)
cout << dBN[k]<<" ";
cout << endl;
cout <<"BNA sgn: "<< sgn << endl;
int sBits;                          // BigNum size in Bits
Ipp32u* dBN1=new Ipp32u[sBN];

ippsExtGet_BN(&sgn,&sBits,dBN1,BNA);// get sgn, size in Bits
cout << "From ExtGet_BN: "<<endl;   // and data
cout << "BNA data: ";
for(int k=0;k<sBN;k++)     // one can replace the unwanted
cout << dBN1[k]<<" ";              // parameters by 0 !!!
cout << endl;
cout <<"BNA sgn: "<< sgn << endl;
cout <<"BNA size in bits: "<<dec << sBits << endl;

tBN("BNA: ",BNA);                   // typing BigNum
delete [] (Ipp8u*) BNA;             // freeing memory
delete [] dBN;
delete [] dBN1;
return 0;}
// Output:
// BNA size: 5
// From Get_BN:
// BNA data: 11234567 1abcdeff 11234567 1abcdeff f1223344
// BNA sgn: 1
// From ExtGet_BN:
```

```
// BNA data: 11234567 1abcdeff 11234567 1abcdeff f1223344
// BNA sgn: 1
// BNA size in bits: 160
// BNA: f12233441abcdeff112345671abcdeff11234567
```

Alternatively, we can use the `Ref_BN` function.

```
// Getting sign, bit-size and Ipp32u array from BN structure
// icpc -ipp=crypto 020Ref_BN+.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"  // nBN, tBN

int main(){
using namespace std ;
Ipp32u A[] = {0x11234567,0x1abcdeff,0xf1223344,
                         0x1abcdeff,0xf1223344};
int dS=(sizeof A)/sizeof(Ipp32u);
IppsBigNumState* BNA=nBN(dS,A);
int sBN;
ippsGetSize_BN(BNA,&sBN);
cout << "BNsize: "<< sBN << endl;
IppsBigNumSGN sgn;
Ipp32u* dBN=0;
int sBits;

ippsRef_BN(&sgn,&sBits,&dBN,BNA);
cout <<"sgn: "<< sgn << endl;
cout <<"BN size in bits: "<< dec << sBits << endl;

cout<<"BNA data: "<<hex;
for(int k=0;k<sBN;k++)
cout << dBN[k]<<" ";
cout<<endl;
tBN("BNA: ",BNA);
delete [] (Ipp8u*)BNA;
return 0;}
// Output:
// BNsize: 5
```

```
// sgn: 1
// BN size in bits: 160
// BNA data: 11234567 1abcdeff f1223344 1abcdeff f1223344
// BNA: f12233441abcdefff12233441abcdeff11234567
```

10.4 Typing BigNums

The function tBN types BigNums. It uses an octet string representation of BigNums, the function Get_BN and additionally the function GetOctString_BN which extracts the octet string representations from the BigNum structure.

```
    void tBN (      const char *Msg,                //message
                    IppsBigNumState *BN)        //BN-argument
```

Remark. Similarly as nBN, tBN is not a part of IPP. It is defined to simplify the code examples.

```
void tBN(const char* Msg,const IppsBigNumState* BNR){
using namespace std;
int sBNR;                          // size of BigNum
ippsGetSize_BN(BNR,&sBNR);         // getting size
IppsBigNumSGN sgn;                 // sign of BigNum
Ipp32u* dBNR=new Ipp32u[sBNR];     // BNR data
ippsGet_BN(&sgn,&sBNR,dBNR,BNR);  // getting BNR sign and data
int size=sBNR;
IppsBigNumState* BN=nBN(size,dBNR);// neglecting  sign
Ipp8u* vBN = new Ipp8u [size*4];   // which is typed below
ippsGetOctString_BN(vBN, size*4, BN);
if(Msg)
  cout << Msg;                      // header
cout.fill('0');
cout << hex ;
if(sgn==0) cout<< "-";              // sign
for(int n=0; n<size*4; n++){
  cout.width(2);
  cout <<(int)vBN[n];               // value
}
cout << dec;
cout.fill(' ');
```

```
cout << endl;
delete [] (Ipp8u*) BN;
delete [] vBN;
}
```

10.5 BigNum comparison

One can check if two BigNums are equal or if a BigNum is equal to zero.

```
 IppStatus ippsCmp_BN (IppsBigNumState *BN1,  //input:BN-arg.1
                       IppsBigNumState *BN,   //input:BN-arg.2
                       Ipp32u *Rsult)          //output:result
  //BN1 == BN2,  0:equal, 1:first greater, 2:second greater
```

```
// BN Comparison, BN introduced as Ipp32u arrays
// icpc -ipp=crypto 023Cmp_BN+.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, tBN

int main(){
   using namespace std;
   const Ipp32u A[] = {0x01234567,0x9abcdeff,0x11223344};
   const Ipp32u B[] = {0x01234566,0x9abcdeff,0x11223344};
   IppsBigNumState* BNA = nBN(sizeof(A)/sizeof(Ipp32u),A);
   IppsBigNumState* BNB = nBN(sizeof(B)/sizeof(Ipp32u),B);
   tBN("A:\n", BNA);
   tBN("B:\n", BNB);
   Ipp32u Rsult;
   ippsCmp_BN(BNA, BNB, &Rsult); // BNA==BNB?
   cout <<  Rsult << " (0:equal; 1:first greater;
                                2:second greater)";
   cout << "\n";
   delete [] (Ipp8u*)BNA;
   delete [] (Ipp8u*)BNB;
   return 0;}

// Output:
```

```
// A:
// 112233449abcdeff01234567
// B:
// 112233449abcdeff01234566
// 1 (0:equal; 1:first greater; 2:second greater)
```

If the `BigNums` in octet string representation are given then the code can be modified as follows:

```
// BN Comparison, BN introduced as Octet Strings
// icpc -ipp=crypto 023Cmp_BNb+.cpp toolb.cpp
#include "ippcp.h"
#include<iostream>
#include "toolb.h" // nBN, tBN

int main(){
using namespace std;
Ipp8u A[] = "\x12\x34\x56\x78\x98\x76\x54\x34";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BNA = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BNA);
tBN("A:\n", BNA);
Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x31";
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BNB = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BNB);
tBN("B:\n", BNB);
Ipp32u Rsult;
ippsCmp_BN(BNA, BNB, &Rsult); // BNA==BNB?
cout <<Rsult<<" (0:equal; 1:first greater; 2:second greater)";
cout << "\n";
delete [] (Ipp8u*)BNA;
delete [] (Ipp8u*)BNB;
return 0;}

// Output:
// A:
// 1234567898765434
// B:
```

```
// 1234567898765431
// 1 (0:equal; 1:first greater; 2:second greater)
```

If one of BigNums is zero one can use the specialized function CmpZero_BN.

```
 IppStatus ippsCmpZero_BN (IppsBigNumState *BN, //input:BN-arg.
                           Ipp32u *Rsult)       //output:result

  //BN == 0,  0:equal, 1:first greater, 2:second greater
```

```
// Comparing BN to Zero
// icpc -ipp=crypto 024CmpZero_BN+.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, tBN

int main(){
using namespace std;
Ipp8u A[] = "\x0";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BNA = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BNA);
tBN("A:\n", BNA);
Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x31";

int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BNB = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BNB);
tBN("B:\n", BNB);
Ipp32u Rsult;
ippsCmpZero_BN(BNA, &Rsult);
cout <<  "A compared to zero: "<<Rsult;
                  // 0:equal 1:first greater 2:second greater
cout << "\n";
ippsCmpZero_BN(BNB, &Rsult);
cout <<  "B compared to zero: "<<Rsult;
                 // 0:equal 1:first greater 2:second greater
cout << "\n";
delete [] (Ipp8u*)BNA;
delete [] (Ipp8u*)BNB;
```

```
return 0;}

// Output:
// A:
// 00000000
// B:
// 1234567898765431
// A compared to zero: 0
// B compared to zero: 1
```

10.6 BigNum addition

The addition of two BigNums can be performed with the use of Add_BN

```
IppStatus ippsAdd_BN (IppsBigNumState *BN1,  //input:BN-arg.1
                      IppsBigNumState *BN2,  //input:BN-arg.2
                      IppsBigNumState *BN3)  //outp:BN-result

                      //BN3 = BN1+BN2
```

```
// Addition of two BN introduced as Ipp32u arrays
// icpc -ipp=crypto 025Add_BN0.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
// BigNum A
const Ipp32u A[] = {0x07654321,0x12345678,0x0fedcba1};
IppsBigNumState* BNA = nBN(sizeof(A)/sizeof(Ipp32u), A);
// BigNum B
const Ipp32u B[] = {0x0fedcba1,0x07654321,0x12345678};
IppsBigNumState* BNB = nBN(sizeof(B)/sizeof(Ipp32u), B);
// BigNum R
int sizeR = IPP_MAX(sizeof(A), sizeof(B));
IppsBigNumState* BNR = nBN(1+sizeR/sizeof(Ipp32u));
// R = A+B
ippsAdd_BN(BNA, BNB, BNR); // BNR=BNA+BNB
// typing A,B,R
```

```
tBN("A=:\n", BNA);
tBN("B=:\n", BNB);
tBN("R=A+B:\n", BNR);
delete [] (Ipp8u*)BNA;
delete [] (Ipp8u*)BNB;
delete [] (Ipp8u*)BNR;

return 0;}

// Output:
// A=:
// 0fedcba11234567807654321
// B=:
// 12345678076543210fedcba1
// R=A+B:
// 2222221919999999917530ec2

// Python:
// A=0x0fedcba11234567807654321
// B=0x12345678076543210fedcba1
// hex(A+B)
// '2222221919999999917530ec2'
```

Now let us consider BigNums in octet string representation.

```
// Addition of two BN introduced as Octet Strings
// icpc -ipp=crypto 025Add_BNb+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
Ipp8u A[] = "\x12\x34\x56\x78\x98\x76\x54\x32";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x32";
```

```
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BN2);
tBN("B:\n", BN2);

int size3 = IPP_MAX(size1, size2);
IppsBigNumState* BN3 = nBN(size3+1);
ippsAdd_BN(BN1, BN2, BN3); // BN3=BN1+BN2
tBN("A+B:\n", BN3);
delete [] (Ipp8u*)BN1;
delete [] (Ipp8u*)BN2;
delete [] (Ipp8u*)BN3;
return 0;}

// Output:
// A:
// 1234567898765432
// B:
// 1234567898765432
// A+B:
// 2468acf130eca864

// Python
// A=0x1234567898765432
// B=A
// hex(A+B)
// '0x2468acf130eca864'
```

10.7 BigNum subtraction

The subtraction of two BigNums can be performed with the use of Sub_BN

```
IppStatus ippsSub_BN (IppsBigNumState *BN1,  //input:BN-arg.1
                      IppsBigNumState *BN2,  //input:BN-arg.2
                      IppsBigNumState *BN3)  //outp:BN-result

                      //BN3 = BN1-BN2
```

```
// Subtraction of two BN introduced as Ipp32u arrays
```

```
// icpc -ipp=crypto 026Sub_BN+.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, tBN

int main(){
using namespace std;
const Ipp32u A[] = {0x01234567,0x9abcdeff,0x11223344};
IppsBigNumState* BNA = nBN(sizeof(A)/sizeof(Ipp32u), A);
const Ipp32u B[] = {0x76543210,0xfedcabee,0xf4332211};
IppsBigNumState* BNB = nBN(sizeof(B)/sizeof(Ipp32u), B);
int sizeR = IPP_MAX(sizeof(A), sizeof(B));
IppsBigNumState* BNR = nBN(sizeR/sizeof(Ipp32u));

ippsSub_BN(BNA, BNB, BNR);   // R=A-B
tBN("A=:\n", BNA);
tBN("B=:\n", BNB);

int BNRsize;
ippsGetSize_BN(BNR,&BNRsize);
IppsBigNumSGN sgn;
Ipp32u* BNRdata=new Ipp32u[BNRsize];
ippsGet_BN(&sgn,&BNRsize,BNRdata,BNR);

cout<<"R=A-B as Ipp32u array: "<< hex;
if(sgn==0) cout<<"-";
for(int k=0;k<BNRsize;k++)
cout << BNRdata[k]<<" ";
cout << endl;
cout<<"R=A-B as Octet String: ";
tBN("",BNR);

delete [] (Ipp8u*)BNA;
delete [] (Ipp8u*)BNB;
delete [] (Ipp8u*)BNR;
delete [] (Ipp32u*)BNRdata;
return 0;}
```

```
// Output:
// A=:
// 112233449abcdeff01234567
// B=:
// f4332211fedcabee76543210
// R=A-B as Ipp32u array: -7530eca9 641fccef e310eecd
// R=A-B as Octet String: -e310eecd641fccef7530eca9

// Python:
// A=0x112233449abcdeff01234567
// B=0xf4332211fedcabee76543210
// hex(A-B)
// '-0xe310eecd641fccef7530eca9L'
```

Now the version with octet string representation.

```
// Subtraction of two BN introduced as Octet Strings
// icpc -ipp=crypto 026Sub_BNb+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
Ipp8u A[] = "\x12\x34\x56\x78\x98\x76\x54\x32";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x33";
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BN2);
tBN("B:\n", BN2);

int size3 = IPP_MAX(size1, size2);
IppsBigNumState* BN3 = nBN(size3);
ippsSub_BN(BN1, BN2, BN3); // BN3=BN1-BN2
tBN("A-B:\n", BN3);
delete [] (Ipp8u*)BN1;
```

```
delete [] (Ipp8u*)BN2;
delete [] (Ipp8u*)BN3;
return 0;}

// Output:
// A:
// 1234567898765432
// B:
// 1234567898765433
// A-B:
// -00000001
```

10.8 BigNum multiplication

The multiplication of two BigNums can be performed with the use of Mul_BN

```
IppStatus ippsMul_BN (IppsBigNumState *BN1,  //input:BN-arg.1
                      IppsBigNumState *BN2,  //input:BN-arg.2
                      IppsBigNumState *BN3)  //outp:BN-result

                      //BN3 = BN1*BN2
```

```
// Multiplication of two BNs introduced as Ipp32u arrays
// icpc -ipp=crypto 027Mul_BN+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
const Ipp32u A[] = {0x01234567,0x9abcdeff,0x11223344};
IppsBigNumState* bnA = nBN(sizeof(A)/sizeof(Ipp32u), A);
const Ipp32u B[] = {0x76543210,0xfedcabee,0x44332211};
IppsBigNumState* bnB = nBN(sizeof(B)/sizeof(Ipp32u), B);
int sizeR = sizeof(A)+sizeof(B);
IppsBigNumState* bnR = nBN(sizeR/sizeof(Ipp32u));
ippsMul_BN(bnB, bnA, bnR);              // R=A*B
tBN("A:\n", bnA);
tBN("B:\n", bnB);
tBN("R=A*B:\n", bnR);
```

```
delete [] (Ipp8u*)bnA;
delete [] (Ipp8u*)bnB;
delete [] (Ipp8u*)bnR;
return 0;}

// Output:
// A:
// 112233449abcdeff01234567
// B:
// 44332211fedcabee76543210
// R=A*B:
// 049081b62ee836bb73b58874cee2591dba38b27b358e7470

// Python:
// A=0x112233449abcdeff01234567
// B=0x44332211fedcabee76543210
// hex(A*B)
// '0x49081b62ee836bb73b58874cee2591dba38b27b358e7470L'
```

Now the version with octet string representation.

```
// Multiplication of two BNs introduced as Octet Strings
// icpc -ipp=crypto 027Mul_BNb+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
using namespace std;
Ipp8u A[] ="\x12\x34\x56\x78\x98\x76\x54\x32";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x32";
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BN2);
tBN("B:\n", BN2);
```

```
int size3 = size1+size2;
IppsBigNumState* BN3 = nBN(size3);
ippsMul_BN(BN1, BN2, BN3);          // BN3=BN1*BN2
tBN("A*B:\n", BN3);
delete [] (Ipp8u*)BN1;
delete [] (Ipp8u*)BN2;
delete [] (Ipp8u*)BN3;
return 0;}

// Output:
// A:
// 1234567898765432
// B:
// 1234567898765432
// A*B:
// 014b66dc33a3d328069d418d19c8d9c4

//Python
//A=0x1234567898765432
//B=0x1234567898765432
//hex(A*B)
//'14b66dc33a3d328069d418d19c8d9c4'
```

Here is an example with a negative factor.

```
// Multiplication of two BNs introduced as Ipp32u arrays
// with opposite signs
// icpc -ipp=crypto 027Mul_BNneg+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, nBNneg, tBN

int main(){
using namespace std;
const Ipp32u A[] = {0x01234567,0x9abcdeff,0x11223344};
IppsBigNumState* bnA = nBN(sizeof(A)/sizeof(Ipp32u), A);
const Ipp32u B[] = {0x76543210,0xfedcabee,0x44332211};
IppsBigNumState* bnB = nBNneg(sizeof(B)/sizeof(Ipp32u), B);
IppsBigNumSGN sgn;                                  //B -negative
```

```
int sizeR = sizeof(A)+sizeof(B);
IppsBigNumState* bnR = nBN(sizeR/sizeof(Ipp32u));
ippsMul_BN(bnB, bnA, bnR); // R=A*B
tBN("A:\n", bnA);
tBN("B:\n", bnB);
tBN("R=A*B:\n", bnR);
delete [] (Ipp8u*)bnA;
delete [] (Ipp8u*)bnB;
delete [] (Ipp8u*)bnR;
return 0;}

// Output:
// A:
// 112233449abcdeff01234567
// B:
// -44332211fedcabee76543210
// R=A*B:
// -049081b62ee836bb73b58874cee2591dba38b27b358e7470

// Python:
// A=0x112233449abcdeff01234567
// B=-0x44332211fedcabee76543210
// hex(A*B)
// '-0x49081b62ee836bb73b58874cee2591dba38b27b358e7470L'
```

There is also the possibility to accumulate the product of two BigNums with third BigNum.

```
// Multiplication of two BNs introduced as Ipp32u arrays
// and accumulation with third BN
// icpc -ipp=crypto 028MAC_BN_I.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
const Ipp32u A[] = {0x01234567,0x9abcdeff,0x11223344};
IppsBigNumState* bnA = nBN(sizeof(A)/sizeof(Ipp32u), A);
const Ipp32u B[] = {0x76543210,0xfedcabee,0x44332211};
IppsBigNumState* bnB = nBN(sizeof(B)/sizeof(Ipp32u), B);
```

```
int sizeR = sizeof(A)+sizeof(B);
Ipp32u R[sizeR];
for(int k=1;k<sizeR;k++) R[k]=0x0;
R[0]=0x1;
IppsBigNumState* bnR = nBN(sizeR/sizeof(Ipp32u),R);
tBN("A:\n", bnA);
tBN("B:\n", bnB);
tBN("R:\n", bnR);
ippsMAC_BN_I(bnB, bnA, bnR); // R=R+A*B
tBN("R=R+A*B:\n", bnR);

delete [] (Ipp8u*)bnA;
delete [] (Ipp8u*)bnB;
delete [] (Ipp8u*)bnR;
return 0;}

// Output:
// A:
// 112233449abcdeff01234567
// B:
// 44332211fedcabee76543210
// R:
// 00000001
// R=R+A*B:
// 049082162ee8370b73b588c4cee2591dba38b27b358e7471

// Python:
// A=0x112233449abcdeff01234567
// B=0x44332211fedcabee76543210
// R=0x1
// hex(R+A*B)
// '0x49081b62ee836bb73b58874cee2591dba38b27b358e7471L'
```

10.9 BigNum division

If two BigNums are given, then their quotient and remainder can be evaluated using function Div_BN.

```
IppStatus ippsDiv_BN (IppsBigNumState *BNA,   //input:BN-arg.1
                      IppsBigNumState *BNB,   //input:BN-arg.2
                      IppsBigNumState *BNQ,  //out:BN-quotient
                      IppsBigNumState *BNR) //out:BN-remainder

                      //BNA = BNQ*BNB + BNR
```

```
// Division with remainder of two BNs A,B introduced as
// Ipp32u arrays
// The result: two BNs Q-quotient, R-remainder
// A=Q*B+R
// icpc -ipp=crypto 029Div_BN+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
const Ipp32u A[] = {0x76543210,0xfedcabee,0x44332211};
IppsBigNumState* bnA = nBN(sizeof(A)/sizeof(Ipp32u), A);
const Ipp32u B[] = {0x01234567,0x9abcdeff,0x11223344};
IppsBigNumState* bnB = nBN(sizeof(B)/sizeof(Ipp32u), B);
int sizeQ = sizeof(A)-sizeof(B)+sizeof(Ipp32u);
IppsBigNumState* bnQ = nBN(sizeQ/sizeof(Ipp32u));
int sizeR = sizeof(B);
IppsBigNumState* bnR = nBN(sizeR/sizeof(Ipp32u));
ippsDiv_BN(bnA, bnB, bnQ, bnR); // A=Q*B+R
tBN("A:\n", bnA);
tBN("B:\n", bnB);
tBN("Q:\n", bnQ);
tBN("R:\n", bnR);
delete [] (Ipp8u*)bnA;
delete [] (Ipp8u*)bnB;
delete [] (Ipp8u*)bnQ;
delete [] (Ipp8u*)bnR;

return 0;}

// Output:
// A:
// 44332211fedcabee76543210
```

```
// B:
// 112233449abcdeff01234567
// Q:
// 00000003
// R:
// 10cc88442ea60ef172ea61db

// Python:
// A=0x44332211fedcabee76543210
// B=0x112233449abcdeff01234567
// hex(A//B)
// '0x3L'
// hex(A%B)
// '0x10cc88442ea60ef172ea61dbL'
```

Now, the version with octet string representation.

```
// Division with remainder of two BNs A,B introduced as
// Octet Strings
// The result: two BNs Q-quotient, R-remainder
// A=Q*B+R
// icpc -ipp=crypto 029Div_BNb+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
Ipp8u A[] = "\x12\x34\x56\x78\x98\x76\x54\x32";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x32";
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BN2);
tBN("B:\n", BN2);

int size3 = size1-size2+1;
```

```
IppsBigNumState* BN3 = nBN(size3);

int size4 = size2;
IppsBigNumState* BN4 = nBN(size4);

ippsDiv_BN(BN1, BN2, BN3, BN4);   //  A=Q*B+R
tBN("Q:\n", BN3);
tBN("R:\n", BN4);
delete [] (Ipp8u*)BN1;
delete [] (Ipp8u*)BN2;
delete [] (Ipp8u*)BN3;
delete [] (Ipp8u*)BN4;
return 0;}

// Output:
// A:
// 1234567898765432
// B:
// 1234567898765432
// Q:
// 00000001
// R:
// 00000000
```

10.10 Reduction modulo BigNum

When two BigNums BN1 and BN2 are given, then the function Mod_BN allows for computing BN3 equal to BN1 mod BN2.

```
IppStatus ippsMod_BN (IppsBigNumState *BN1,  //input:BN-arg.1
                      IppsBigNumState *BN2,  //input:BN-arg.2
                      IppsBigNumState *BN3)  //outp:BN-result

                      //BN3 = BN1 mod BN2
```

```
// Modular reduction of BN1 with respect modulus BN2
// BN1,BN2 introduced as Octet Strings
// BN3 = BN1 mod BN2
```

```
// icpc -ipp=crypto 030Mod_BN+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h"                  // nBN, tBN

int main(){
Ipp8u A[] = "\x12\x34\x56\x78\x98\x76\x54\x32";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u M[] = "\x12\x34\x56\x78\x98";
int size2 = (sizeof(M)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(M, sizeof(M)-1, BN2);
tBN("M:\n", BN2);

int size3 = size2;
IppsBigNumState* BN3 = nBN(size3);
ippsMod_BN(BN1, BN2, BN3);         // BN3 = BN1 mod BN2
tBN("A mod M:\n", BN3);
delete [] (Ipp8u*)BN1;
delete [] (Ipp8u*)BN2;
delete [] (Ipp8u*)BN3;
return 0;}

// Output:
// A:
// 1234567898765432
// M:
// 0000001234567898
// A mod M:
// 00765432

// Python:
// A=0x1234567898765432
// M=0x1234567898
// hex(A%M)
```

```
// '0x765432'
```

10.11 Gcd for BigNums

The greatest common divisor of two BigNums can be computed using the function ippsGcd_BN.

```
IppStatus ippsGcd_BN (IppsBigNumState *BN1,  //input:BN-arg.1
                      IppsBigNumState *BN2,  //input:BN-arg.2
                      IppsBigNumState *BN3)  //outp:BN-result

                      //BN3=Gcd(BN1,BN2)
```

```
// Greatest common divisor of two BNs
// BN3=Gcd(BN1,BN2)
// icpc -ipp=crypto 031Gcd_BN+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h"               // nBN, tBN

int main(){
Ipp8u A[] = "\x32\x34\x56\x78\x98\x76\x54\x32";
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x32";
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BN2);
tBN("B:\n", BN2);

int size3 = IPP_MAX(size1, size2);
IppsBigNumState* BN3 = nBN(size3);
ippsGcd_BN(BN1, BN2, BN3);      // BN3=Gcd(BN1,BN2)
tBN("gcd(A,B):\n", BN3);
delete [] (Ipp8u*)BN1;
delete [] (Ipp8u*)BN2;
```

```
delete [] (Ipp8u*)BN3;
return 0;}

// Output:
// A:
// 3234567898765432
// B:
// 1234567898765432
// gcd(A,B):
// 00000002

// sage:
// A=0x3234567898765432
// B=0x1234567898765432
// gcd(A,B)
// 2
```

10.12 Inverse modulo BigNum

When a Bignum modulus M is fixed then by the multiplicative inverse of a BigNum A we mean a BigNum B such that A*B=1 mod M. To obtain such a multiplicative inverse we can use the function ModInv_BN.

```
IppStatus ippsModInv_BN( IppsBigNumState *BN1,  //inp:BN-arg.1
                         IppsBigNumState *BN2,  //inp:BN-arg.2
                                                   //-modulus
                         IppsBigNumState *BN3) //out:BN-result

 //BN3=Inverse(BN1) mod BN2  (BN1*BN3=1 mod BN2)
```

```
// Inverse of BN1 modulo BN2
// BN3 = inverse of BN1 modulo BN2   (BN3*BN1=1 mod BN2)
// icpc -ipp=crypto 032ModInv_BN+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN, tBN

int main(){
Ipp8u A[] = "\x56\x78\x98\x76\x54\x32";
```

```
int size1 = (sizeof(A)-1+3)/4;
IppsBigNumState* BN1 = nBN(size1);
ippsSetOctString_BN(A, sizeof(A)-1, BN1);
tBN("A:\n", BN1);

Ipp8u B[] ="\x12\x34\x56\x78\x98\x76\x54\x31";
int size2 = (sizeof(B)-1+3)/4;
IppsBigNumState* BN2 = nBN(size2);
ippsSetOctString_BN(B, sizeof(B)-1, BN2);
tBN("M:\n", BN2);

int size3 = size2;
IppsBigNumState* BN3 = nBN(size3);
ippsModInv_BN(BN1, BN2, BN3); // BN3 = inverse of BN1 mod BN2
tBN("inv A mod M:\n", BN3);   // (if there exists)
delete [] (Ipp8u*)BN1;
delete [] (Ipp8u*)BN2;
delete [] (Ipp8u*)BN3;
return 0;}

// Output:
// A:
// 0000567898765432
// M:
// 1234567898765431
// inv A mod M:
// 095b097ec4da476e

// sage:
// A=0x567898765432
// M=0x1234567898765431
// hex(inverse_mod(A,M))
// '95b097ec4da476e'
```

10.13 The idea of Montgomery reduction

The usual way of exponentiation modulo N consists of successive repeating the operations of multiplying and division by N with remainder. Mont-

gomery's approach allows for avoiding the intermediate division operations (which are expensive). In this approach the so-called Montgomery multiplication is used. To multiply two numbers, they are first transformed to Montgomery representation, the results are multiplied in Montgomery sense and the product is represented back in the normal form. For two factors, this takes longer than the normal multiply and division with remainder procedure, but for exponentiation, where we would Montgomery-multiply many times, we only have to do the representation change at the beginning and at the end (the intermediate divisions are avoided).
The basic difficulty in Montgomery approach is to keep the intermediate products "small" without computing their remainders mod N. The Montgomery reduction is the operation that makes it possible.
To show how the Montgomery method works, we introduce two operations $MontForm$ and $MontMul$. Assume that an integer $R > N$ is given, such that gcd(R, N)=1. For integers x, y, such that $0 \leq x, y < N$ we define

$$MontForm(x) = (x \cdot R) \bmod N,$$

$$MontMul(x, y) = (x \cdot y \cdot R^{-1}) \bmod N.$$

The second formula contains the reduction step. In practical algorithms, the division by R is replaced by faster operations which give results equivalent modulo N.

Example. Computing x^5 mod N with Montgomery method.

Montgomery operation	Equiv. mod N	Result mod N
$a_1 = MontMForm(x)$	xR	xR
$a_2 = MontMul(a_1, a_1)$	$xR\,xR\,R^{-1}$	x^2R
$a_4 = MontMul(a_2, a_2)$	$x^2R\,x^2R\,R^{-1}$	x^4R
$a_5 = MontMul(a_4, a_1)$	$x^4R\,xR\,R^{-1}$	x^5R
$x_5 = ModMul(a_5, 1)$	$x^5R \cdot 1 \cdot R^{-1}$	x^5

10.14 Montgomery reduction in IPP

IPP contains seven functions related to Montgomery reduction.
All functions use the `IppsMontState` context. To obtain the size of the `IppsMontState` context one can use the function `MontGetSize`.

```
IppStatus ippsMontGetSize(IppsExpMethod meth, //input:meth.of
                                                //exponentiation
                          int Modsize,    //input:modulus size
                                             //in Ipp32u words
                          int* MontCsize) //output:size of the
                                        //IppsMontState context
```

The `IppsMontState` context can be initialized with the help of `MontInit`.

```
   IppStatus ippsMontInit(IppsExpMethod meth, //input:meth.of
                                                //exponentiation
                        int Modsize,      //input:modulus size
                                             //in Ipp32u words
                        IppsMontState* Mont) //out:initialized
                                        //IppsMontState context
```

The value of the modulus can be set with the help of `MontSet`.

```
IppStatus ippsMontSet(const Ipp32u* M ,       //input:  value
                                               //of the modulus
                          int Modsize,    //input:modulus size
                                             //in Ipp32u words
                          IppsMontState* Mont) //IppsMontState
                                                  // context
```

The value of the modulus can be extracted from Montgomery state context using `MontGet`.

```
  IppStatus ippsMontGet(      Ipp32u* M ,       //output:value
                                              //of the modulus
                         int* Modsize,         //modulus size
                                             //in Ipp32u words
                         IppsMontState* Mont) //IppsMontState
                                                  // context
```

To convert a `BigNum` to Montgomery form one can use the function `MontForm`.

```
IppStatus ippsMontForm(IppsBigNumState* A ,  //input:BigNum A
                       IppsMontState* Mont,  //input:modulus
                       IppsBigNumState* r) //output:BigNum r
                                            // r=A*R mod M
```

The Montgomery multiplication can be performed with the help of `MontMul`.

```
  IppStatus ippsMontMul(IppsBigNumState* A , //input:BigNum A
                       IppsBigNumState* B,  //input:BigNum B
                       IppsMontState* Mont,  //input:modulus
                       IppsBigNumState* R) //output:BigNum R
                                             //R=A*B mod M
```

```
// Montgomery  multiplication of two BNs modulo third BN
// R = A*B mod M
// icpc -ipp=crypto 033MontMul+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h" // nBN,tBN
#include<iostream>
int main(){
   using namespace std;
   const Ipp32u M[] = {0x01234567,0x9abcdeff,0x11223344};
   IppsBigNumState* bnM = nBN(sizeof(M)/sizeof(Ipp32u), M);
   int size1;
   ippsMontGetSize(IppsBinaryMethod, sizeof(M)/sizeof(Ipp32u),
                                                    &size1);
   IppsMontState* Mont = (IppsMontState*)( new Ipp8u [size1]);
   ippsMontInit(IppsBinaryMethod, sizeof(M)/sizeof(Ipp32u),
                                                      Mont);
   ippsMontSet(M, sizeof(M)/sizeof(Ipp32u), Mont);
   tBN("M:\n", bnM);
   int Msize=sizeof(M)/sizeof(Ipp32u);
   Ipp32u N[Msize];
   int Mlen;
   ippsMontGet(N,&Mlen, Mont);
   cout <<"Modulus from MontGet: "<<endl;
   for(int k=0;k<Mlen;k++)
   cout <<hex<< N[k]<<" ";
   cout<<endl;
```

```
   const Ipp32u A[] = {0x00134567,0x9abcdeff,0x01223344};
   IppsBigNumState* bnA = nBN(sizeof(A)/sizeof(Ipp32u),A);
   tBN("A:\n", bnA);
   ippsMontForm(bnA, Mont, bnA);  // A in Montgomery form
   const Ipp32u B[] = {0x00000067,0x9abcdeff,0x11111111};
   IppsBigNumState* bnB = nBN(sizeof(B)/sizeof(Ipp32u), B);
                                        // B in usual form
   tBN("B:\n", bnB);
   IppsBigNumState* bnR = nBN(sizeof(M)/sizeof(Ipp32u));
   ippsMontMul(bnA, bnB, Mont, bnR);    // R=MontMul(A,B)
   tBN("R = A*B mod M:\n", bnR);
   delete [] (Ipp8u*)bnM;
   delete [] (Ipp8u*)Mont;
   delete [] (Ipp8u*)bnA;
   delete [] (Ipp8u*)bnB;
   delete [] (Ipp8u*)bnR;
return 0;}

//M:
//112233449abcdeff01234567
//Modulus from MontGet:
//1234567 9abcdeff 11223344
//A:
//012233449abcdeff00134567
//B:
//111111119abcdeff00000067
//R = A*B mod M:
//02fbf4bbeb0ac9c241323ddb


//Python:
//M=0x112233449abcdeff01234567
//A=0x012233449abcdeff00134567
//B=0x111111119abcdeff00000067
//hex((A*B)%M)
//'2fbf4bbeb0ac9c241323ddb'
```

The Montgomery exponentiation is implemented in function `MontExp`.

```
 IppStatus ippsMontExp(IppsBigNumState* A,    //input:BigNum A
                                            //in Montgomery form
                       IppsMontState* Mont,  //input:modulus
                       IppsBigNumState* R) //output:BigNum R
                                        // R=A^E*R^(-(E-1)) mod M
        //If A=MontForm(a), then a^E = MontMul(R,1) mod M
```

Remark. The last comment follows from the fact that if $A = aR \bmod M$ and $R = A^E R^{-(E-1)} \bmod M$, then $MontMul(R, 1) = A^E R^{-(E-1)} R^{-1} = A^E R^{-E} = a^E \bmod M$.

```
// Montgomery exponentiation
// R = A^B mod N  (A in Montgomery form)
// icpc -ipp=crypto 034MontExp+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h"             // nBN, tBN

int main(){
   int size;
   // N
   Ipp32u N[] = {0x1111a009};
   ippsMontGetSize(IppsBinaryMethod, sizeof(N)/sizeof(Ipp32u),
                                                        &size);
   IppsMontState* Mont = (IppsMontState*)( new Ipp8u [size] );
   ippsMontInit(IppsBinaryMethod, sizeof(N)/sizeof(Ipp32u),
                                                         Mont);
   ippsMontSet(N, sizeof(N)/sizeof(Ipp32u), Mont);
   // A
   Ipp32u A[] = {0x2};
   IppsBigNumState* bnA = nBN(sizeof(A)/sizeof(Ipp32u), A);
   tBN("A:\n", bnA);

   ippsMontForm(bnA, Mont, bnA);      // A  in Montgomery form
   tBN("A in Mont form:\n", bnA);
   // exponent B
   Ipp32u B[] = {0x80};
   IppsBigNumState* bnB = nBN(sizeof(B)/sizeof(Ipp32u), B);
                                           // B in usual form
   tBN("B:\n", bnB);
```

```
   //  R = A^B mod N
   IppsBigNumState* bnR = nBN(sizeof(N)/sizeof(Ipp32u));
   ippsMontExp(bnA, bnB, Mont, bnR);      // R = MontExp(A,B)
   Ipp32u one[] = {0x1};
   IppsBigNumState* bn1 = nBN(sizeof(one)/sizeof(Ipp32u), one);
   ippsMontMul(bnR, bn1, Mont, bnR);      // R = MontMul(R,1)
   tBN("R = A^B mod N:\n", bnR);

  delete [] (Ipp8u*)Mont;
  delete [] (Ipp8u*)bnA;
  delete [] (Ipp8u*)bnB;
  delete [] (Ipp8u*)bnR;
  delete [] (Ipp8u*)bn1;

return 0;}

//Output:
//A:
//00000002
//A in Mont form:
//1100defb
//B:
//00000080
//R = A^B mod N:
//06e97778

//Python:
//N=0x1111a009
//A=0x2
//B=0x80
//hex(pow(A,B,N))
//'0x6e97778'
```

10.15 Pseudorandom generator set-up

A general description of pseudorandom generators can be found in [HAC], chapter 5.
To set a pseudorandom number generator in IPP, one performs the follow-

ing steps.
1. The `IppsPRNGState` context size is evaluated with the help of `PRNGGetSize`.

```
IppStatus ippsPRNGGetSize( int* CtxSize) //output:the size of
                              //IppsPRNGState context in bytes
```

2. After allocating sufficient memory for `IppsPRNGState` context, it is initialized with the help of `PRNGInit`.

```
   IppStatus ippsPRNGInit( int sBits,      //input:Bit-size of
             // the seed from 1 to 512 bits, default- 160-bits
                  IppsPRNGState* Ctx) //IppsPRNGState* context
```

3. The values of the seed, entropy augment, modulus and the initial hash value of the SHA-1 algorithm for the pseudorandom generator can be set using: `PRNGSetSeed`, `PRNGSetAugment`, `PRNGSetModulus`, `PRNGSetH0`.

```
IppStatus ippsPRNGSetSeed(const IppsBigNumState* seed, //input:
                                                  // BigNum-seed
                   IppsPRNGState* Ctx) //IppsPRNGState* context
                              // The default seed value is 0x0
```

```
 IppStatus ippsPRNGSetAugment( const IppsBigNumState* augment,
                                     // input:BigNum-augment
                   IppsPRNGState* Ctx) //IppsPRNGState* context
  //the augment is of sBits length, default augment val.is 0x0
```

```
    IppStatus ippsPRNGSetModulus( const IppsBigNumState* mod,
                                     // input:BigNum-modulus
                   IppsPRNGState* Ctx) //IppsPRNGState* context
      // modulus is up to 160-bits, default modulus value is
                   // 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
  IppStatus ippsPRNGSetH0( const IppsBigNumState* h0, //input:
                                    //BigNum-initial hash value
                  IppsPRNGState* Ctx) //IppsPRNGState* context
        // h0 is up to 160-bits, default initial hash value is
                 // 0xC3D2E1F01032547698BADCFEEFCDAB8967452301
```

10.16 Pseudorandom `BigNum` generation

The function `PRNGGen_BN` can be used to generate pseudorandom `BigNums`.

```
 IppStatus ippsPRNGGen_BN(IppsBigNumState* randBN,  //output:
                                          //pseudorandom BigNum
                             int Bits,        // randBN size
                             void* Rnd)      //rand.bit.gen.
```

The function `rand()` from the standard library is not sufficiently secure for cryptographic applications but can be used to prepare auxiliary arrays of Ipp32u random integers.

```
Ipp32u* rand32(Ipp32u* X, int size)
{
   for(int n=0; n<size; n++)
      X[n] = (rand()<<16) + rand();
   return X;
}
```

Using the functions from the previous subsection one can define new pseudorandom generators. For simplicity we use the default values of modulus, initial hash and entropy augment.

```
IppsPRNGState* nPRNG(int sBits)
{
   int sSize =(sBits+31)>>5;
   Ipp32u* seed = new Ipp32u [sSize];
//   Ipp32u* augm = new Ipp32u [sSize];
//   Ipp32u* h0   = new Ipp32u [sSize];
//   Ipp32u* mod   = new Ipp32u [sSize];
   int bnSize;
```

```
   IppsBigNumState* Tmp;
   ippsPRNGGetSize(&bnSize);
   IppsPRNGState* Ctx = (IppsPRNGState*)( new Ipp8u [bnSize] );
   ippsPRNGInit(sBits, Ctx);
   ippsPRNGSetSeed(Tmp=nBN(sSize,rand32(seed,sSize)), Ctx);
//   ippsPRNGSetAugment(Tmp=nBN(sSize,rand32(augm,sSize)),Ctx);
//   delete [] (Ipp8u*)Tmp;
//   ippsPRNGSetH0(Tmp=nBN(sSize,rand32(h0,sSize)),Ctx);
//   delete [] (Ipp8u*)Tmp;
//   ippsPRNGSetModulus(Tmp=nBN(sSize,rand32(mod,sSize)),Ctx);
   delete [] (Ipp8u*)Tmp;
   delete [] seed;
//   delete [] augm;
//   delete [] h0;
//   delete [] mod;
   return Ctx;
}
```

The function we have just defined can be used to generate pseudorandom BigNums as follows.

```
// Generating random BN with ippsPRNGen_BN and nPRNG function
// icpc -ipp=crypto 035PRNG0+.cpp toolb.cpp
#include "ippcp.h"
#include "toolb.h"               // nPRNG, nBN, tBN
#include<ctime>

int main(){
using namespace std;
srand(time(0));
IppsPRNGState* Rand=nPRNG();      // new PRNG
const int Bits = 1024;
IppsBigNumState* bnX = nBN(Bits/32);
ippsPRNGen_BN(bnX, Bits, Rand);   // bnX - random BN
tBN("X: ", bnX);
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)bnX;
return 0;}
```

```
// Output: (random)
X:
376e985312d49fe648a453927d57504ae3ed6e2b3d08d7a29c896956087ded4c
0b9cbc55594fee8dbddd93944fe162e03ef186f3bcc0adb1dfccbb81e249498d
eab1c7b71419f0f7d8a25f4eaa1f0121d576489f88c51fc83cfbc880d01b211b
c891cdd393472baa74ce542a9bd2eb89654f647bd0034e347b6815b7e2e08072
```

10.17 IPP prime number generator

In this section we present IPP functions for probable prime `BigNum` generation. By the probable prime we mean the number which passes the Miller-Rabin pseudoprimality test. A more detailed description can be found in [HAC], chapter 4.4.
To generate a probable prime in IPP the following steps are needed.
1. Evaluation of the `IppsPrimeState` context size using `PrimeGetSize`.

```
IppStatus ippsPrimeGetSize(int Bits,      //input:size in bits
                                    //of the pseudoprime BigNum
                           int* size)//output:pseudoprime gen.
                                       //context size in bytes
```

2. Memory allocation for `IppsPrimeState` context and its initialization with `PrimeInit`.

```
IppStatus ippsPrimeInit(int Bits,        //input:size in bits
                                   //of the pseudoprime BigNum
                        IppsPrimeState* Ctx)   // initialized
                                     //IppsPrimeState context
```

3. Generation of probable prime with the help of `PrimeGen`.

```
IppStatus ippsPrimeGen(int Bits,         //input:size in bits
          int Trials,          //input:number of Miller tests
          IppsPrimeState* Ctx,//output:IppsPrimeState context
          IppBitSupplier PRNGen,     //input:random generator
          void* Rand)             //input:random gen.context
```

Remark. The value of `IppStatus` can take the value:
`ippStsInsufficientEntropy`, for example one can obtain the output:

`PrimeGen: ippStsInsufficientEntropy: Insufficient entropy in the random seed and stimulus bit string caused the prime/key generation to fail`. In that case the parameters of the pseudorandom generator should be changed and the function `PrimeGet` should be called again.

4. The value of the prime can also be set with the help of `PrimeSet`

```
IppStatus ippsPrimeSet(const Ipp32u* P,  //input:Prime value,
                                                 //Ipp32u array
         int Bits,                  //input:size of the prime
         IppsPrimeState* Ctx) //output:IppsPrimeState context
```

or `PrimeSet_BN`.

```
IppStatus ippsPrimeSet_BN(const IppsBigNumState* P,  //input:
                                          //BigNum prime value
         IppsPrimeState* Ctx) //output:IppsPrimeState context
```

5. The pseudoprimality test is implemented in the function `PrimeTest`.

```
IppStatus ippsPrimeTest(int Trials, //input:num.of prim.tests
         Ipp32u* result,              //output:  test result
         IppsPrimeState* Ctx,  //input:IppsPrimeState context
         IppBitSupplier PRNGen,     //input:random generator
         void* Rand)              //input:random gen.context
```

6. The value of the prime `BigNum` can be obtained using `PrimeGet_BN`.

```
 IppStatus ippsPrimeGet_BN( IppsBigNumState* P,    //output:
                                          //BigNum prime value
    const IppsPrimeState* Ctx) //input:IppsPrimeState context
```

10.18 Prime `BigNum` generation

By analogy with `nPRNG` to create a new pseudoprime generator we can use the function `nPrimG`.

```
IppsPrimeState* nPrimG(int Bits,IppsPRNGState* Rand)
```

```
{
int size;
ippsPrimeGetSize(Bits, &size);
IppsPrimeState* PrimCtx = (IppsPrimeState*)( new Ipp8u [size] );
ippsPrimeInit(Bits, PrimCtx);
return PrimCtx;
}
```

Here is an example how to generate and test pseudoprime `BigNums`.

```
// Prime BN Generation
// icpc -ipp=crypto 036Prime0+.cpp toolb.cpp
#include "ippcp.h"
#include "ippcore.h"
#include <iostream>
#include "toolb.h" // nPRNG, nPrimG, nBN, tBN
#include <ctime>

int main(){
using namespace std;
srand(time(0));
int size;
int BitS = 512;
int nTrials=20;
IppsPRNGState* pRand=nPRNG();
IppsPrimeState* PrimCtx=nPrimG(BitS ,pRand);
IppStatus w;
w=ippsPrimeGen(BitS,nTrials,PrimCtx,ippsPRNGen,pRand);
cout << "PrimeGen: "<<ippGetStatusString(w) << endl;
Ipp32u result;
ippsPrimeTest(nTrials, &result, PrimCtx,ippsPRNGen, pRand);
IS_PRIME==result?
   cout <<"Primality test of P passed\n" :
   cout <<"Primality test of P failed\n";

IppsBigNumState* bnP = nBN(BitS/32);
ippsPrimeGet_BN(bnP,PrimCtx);
tBN("P: ",bnP);
```

```
Ipp32u bnuPrime1[] =
{0x6e1635f5,0x8a14cac0,0x26232c99,0x16eb265a};
IppsBigNumState* bnP1 = nBN(4, bnuPrime1);
ippsPrimeSet_BN(bnP1,PrimCtx);
ippsPrimeTest(50, &result, PrimCtx,ippsPRNGen, pRand);
IS_PRIME==result?
cout <<"Primality test of P1 passed\n" :
 cout <<"Primality test of P1 failed\n";
delete [] (Ipp8u*)pRand;
delete [] (Ipp8u*)bnP;
delete [] (Ipp8u*)bnP1;
delete [] (Ipp8u*)PrimCtx;
return 0;}

//Output:
//PrimeGen: ippStsNoErr: No error, it's OK
//Primality test of P passed
//P:
//f513e29b719bd39e3a3ea471f348951a6eb40f9432750193cf8b02a8a7
//12141b62daf470a72b847ca81999bb586f1de8a04e3849cd4c2a3ddeb7
//fa5e2a64c173
//Primality test of P1 passed
```

Chapter 11

RSA cryptosystem

A general description of the RSA system can be found in [HAC], section 8.2 and [PKCS 1].

11.1 Creating a new RSA context

To start working with RSA we have to prepare an appropriate context.

1. Calculation of RSA context size with the use of `RSAGetSize`.

```
IppStatus ippsRSAGetSize(int NbitS,          //input:N bit-size
                         int PbitS,          //input:P bit-size
                         IppRSAKeyType Ktype,  //input:Key type
                         int* size)        //output:RSA context
                                               //size in bytes
 // Key type can take values:  IppRSApublic or IppRSAprivate
```

2. Allocation of sufficiently large space for the RSA context and its initialization using `RSAInit`.

```
 IppStatus ippsRSAInit(int NbitS,           //input:N bit-size
                       int PbitS,          //input:P bit-size
                       IppRSAKeyType Ktype,  //input:Key type
                       IppRSAState* Ctx) //output:initialized
                                             //RSA context
```

To perform these tasks we shall use the function:

```
IppsRSAState* nRSA(int NbitS, int PbitS, IppRSAKeyType Ktype)
{
   int size;
   ippsRSAGetSize(NbitS,PbitS, Ktype, &size);
   IppsRSAState* Ctx = (IppsRSAState*)( new Ipp8u [size] );
   ippsRSAInit(NbitS,PbitS, Ktype, Ctx);
   return Ctx;
}
```

3. When the RSA context is initialized one can generate RSA keys with the use of the function `RSAGenerate`.

```
IppStatus ippsRSAGenerate(IppsBigNumState* E,  //input:public
                                                  // key exponent
               int NbitS,                  //input:N bit-size
               int PbitS,                  //input:P bit-size
               int Trials,                  //input:number of
                                              //Miller tests
               IppRSAState* Ctx,         //output:RSA context
               IppBitSupplier PRNGen, //input:pseudorand.gen.
               void* Rand)     //input:pseudorand gen.context
```

4. The just generated keys can be validated using `RSAValidate`.

```
IppStatus ippsRSAValidate(IppsBigNumState* E, //input:public
                                                // key exponent
               int Trials,                  //input:numb of
                                             //Miller tests
               Ipp32u* Reslt,     //output:validation result
               IppRSAState* Ctx,         //input:RSA context
               IppBitSupplier PRNGen,//input:pseudorand.gen.
               void* Rand)    //input:pseudorand gen.context
```

5. The keys can be set using `RSASetKey` and extracted from the RSA context using `RSAGetKey`.

```
IppStatus ippsRSAGetKey(IppsBigNumState* Key,    //output:key
                                                 //component value
                        IppsRSAKeyTag tag,  //output:selected
                                            //component of the key
                        IppRSAState* Ctx) //input:RSA context
```

The possible values of the `tag` for `RSAGetKey` and `RSASetKey` are: `IppRSAkeyN`, `IppRSAkeyP`, `IppRSAkeyQ`, `IppRSAkeyE`, `IppRSAkeyD`, `IppRSAkeyDp`, `IppRSAkeyDq`, `IppRSAkeyQinv`.

```
IppStatus ippsRSASetKey(const IppsBigNumState* key,//input:key
                                                 //component value
                        IppsRSAKeyTag tag,    //input:selected
                                            //component of the key
                        IppRSAState* Ctx) //output:RSA context
```

Now we can present some applications of the functions we have described. First let us generate the appropriate keys and check if they are valid.

```
// Generation and validation of RSA keys
// icpc -xHOST  -ipp=crypto  037RSA00.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include "toolb.h" // nBN, nPRNG, tBN, nRSA
#include<iostream>
int main(){
using namespace std;
static Ipp32u dataE[] = {0x11};
IppsBigNumState* E = nBN(sizeof(dataE)/sizeof(dataE[0]),dataE);
int NbitS=1024;
int PbitS=512;
int Trials=20;
IppsPRNGState* Rand = nPRNG();
Ipp32u privres;

IppsRSAState* RSAprv = nRSA(NbitS,PbitS , IppRSAprivate);
ippsRSAGenerate(E,NbitS,PbitS,Trials,RSAprv, ippsPRNGen, Rand);
ippsRSAValidate(E, Trials, &privres, RSAprv, ippsPRNGen, Rand);
if(IS_VALID_KEY!=privres)
{  cout <<"invalid priv. keys" <<endl;
```

```
   return 0;
}

IppsBigNumState* N = nBN(NbitS/32);
IppsBigNumState* P = nBN(PbitS/32);
IppsBigNumState* Q = nBN(PbitS/32);
IppsBigNumState* D = nBN(NbitS/32);

ippsRSAGetKey(N,IppRSAkeyN,RSAprv);
ippsRSAGetKey(P,IppRSAkeyP,RSAprv);
ippsRSAGetKey(Q,IppRSAkeyQ,RSAprv);
ippsRSAGetKey(D,IppRSAkeyD,RSAprv);
tBN("N: ",N);
tBN("P: ",P);
tBN("Q: ",Q);
tBN("D: ",D);

delete [] (Ipp8u*)E;
delete [] (Ipp8u*)RSAprv;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)N;
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)Q;
delete [] (Ipp8u*)D;

return 0;}
/*
N:
99197f0db34ed292e78e89eb9e5259c1a65a27a379514004b474ad5774edee68
dc4a21d7ad045536e218e43dff0bcd51a650fce06e2e05581e13510f14f5dcb8
127af4e510eb90ea95189046139bd5efd021673a4221d0598076c61fc0691195
59f51e09ea6f676b5c1b1a4cda3b78862458257ddf4f20d06acadded03cccb37
P:
ddf492adbd48a07a831d92dd90e7d1f2b37723e59311158fd91edf8d6660cde8
21e71358c291dc5d5e752b3c67fe659f7f5f4a5c7169ec856b6cf2e5593a45c3
Q:
b095316ff598058f340807a626866597d6d904cca481b1ed8f363767b0dee7bd
2b5881d5a03b89eab26d3572cc67977df3b0ce3c72964ac1444b909748d9a97d
```

```
D:
1202ffe37e81be6ba2c579a33fcd73f8aa28b95e86be43c451772373b3674939
835403fb41880a0674f3de9de1c52736c845c365b29c00a0f47abe3e02774724
87fe41f9565ba3298385fecb925fb8482654da2e1f5c1f28f3c7ba414113ecb2
d45197a15b403c5e8151f7b831a0a51b6026981af063a301251137d0fc7019e1
*/
```

11.2 RSA encryption and decryption

The RSA encryption can be performed by the function `RSAEncrypt`.

```
IppStatus ippsRSAEncrypt(const IppsBigNumState* plain, //input:
                                                    //plain text
                      IppsBigNumState* ciph,//output:cipher text
                      IppRSAState* Ctx)      //input:RSA context
```

The corresponding decryption performs `RSADecrypt`.

```
IppStatus ippsRSADecrypt(IppsBigNumState* ciph,         //input:
                                                   //cipher text
                      IppsBigNumState* plain,//output:plain text
                      IppRSAState* Ctx)      //input:RSA context
```

Here is an example how to set the RSA keys manually and how to perform the encryption and decryption. The data are taken from the previous example.

```
// RSA encryption and decryption
// icpc -ipp=crypto 037RSA10.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h"

static Ipp32u dataN[] = {
0x03cccb37,0x6acadded,0xdf4f20d0,0x2458257d,0xda3b7886,0x5c1b1a4c,
0xea6f676b,0x59f51e09,0xc0691195,0x8076c61f,0x4221d059,0xd021673a,
0x139bd5ef,0x95189046,0x10eb90ea,0x127af4e5,0x14f5dcb8,0x1e13510f,
0x6e2e0558,0xa650fce0,0xff0bcd51,0xe218e43d,0xad045536,0xdc4a21d7,
0x74edee68,0xb474ad57,0x79514004,0xa65a27a3,0x9e5259c1,0xe78e89eb,
```

```
0xb34ed292,0x99197f0d
};

static Ipp32u dataP[] = {
0x593a45c3,0x6b6cf2e5,0x7169ec85,0x7f5f4a5c,0x67fe659f,0x5e752b3c,
0xc291dc5d,0x21e71358,0x6660cde8,0xd91edf8d,0x9311158f,0xb37723e5,
0x90e7d1f2,0x831d92dd,0xbd48a07a,0xddf492ad
};

static Ipp32u dataQ[] = {
0x48d9a97d,0x444b9097,0x72964ac1,0xf3b0ce3c,0xcc67977d,0xb26d3572,
0xa03b89ea,0x2b5881d5,0xb0dee7bd,0x8f363767,0xa481b1ed,0xd6d904cc,
0x26866597,0x340807a6,0xf598058f,0xb095316f
};
static Ipp32u dataE[] = {0x11};

int main(){
using namespace std;
IppsBigNumState* P = nBN(sizeof(dataP)/sizeof(dataP[0]),dataP);
IppsBigNumState* Q = nBN(sizeof(dataQ)/sizeof(dataQ[0]),dataQ);
IppsBigNumState* N = nBN(sizeof(dataN)/sizeof(dataN[0]),dataN);
IppsBigNumState* E = nBN(sizeof(dataE)/sizeof(dataE[0]),dataE);
IppsRSAState* RSApub  = nRSA(sizeof(dataN)*8, sizeof(dataP)*8,
                                                IppRSApublic);
IppsRSAState* RSAprv = nRSA(sizeof(dataN)*8, sizeof(dataP)*8,
IppRSAprivate);
IppsBigNumState* Pm1 = nBN(sizeof(dataP)/sizeof(dataP[0]));
IppsBigNumState* Qm1 = nBN(sizeof(dataQ)/sizeof(dataQ[0]));
const Ipp32u One[] = {0x00000001};
IppsBigNumState* ONE = nBN(sizeof(One)/sizeof(Ipp32u), One);
ippsSub_BN(P, ONE, Pm1);
ippsSub_BN(Q, ONE, Qm1);
IppsBigNumState* phi = nBN(sizeof(dataN)/sizeof(dataN[0]));
ippsMul_BN(Pm1, Qm1, phi);
IppsBigNumState* D = nBN(sizeof(dataN)/sizeof(dataN[0]));
ippsModInv_BN(E, phi, D);
IppsPRNGState* Rand = nPRNG();
ippsRSASetKey(E, IppRSAkeyE, RSApub);           //public key
```

```
ippsRSASetKey(N, IppRSAkeyN, RSApub);
ippsRSASetKey(P, IppRSAkeyP, RSAprv);           //private key
ippsRSASetKey(Q, IppRSAkeyQ, RSAprv);
ippsRSASetKey(D, IppRSAkeyD, RSAprv);
Ipp32u result;
ippsRSAValidate(E, 50, &result, RSAprv, ippsPRNGen, Rand);
  if(IS_VALID_KEY!=result)                    //key validation
{   cout <<"invalid keys prv" <<endl;
   return 0;
}
Ipp32u dataM[] = {                            // plain text
0x12345678,0xabcde123,
0x87654321,0x111aaaa2,0xbbbbbbbb,0xcccccccc,0x12237777,0x82234587,
0x1ef392c9,0x43581159,0xb5024121,0xa48D2869,0x2abababa,0x1a2b3c22,
0xa47728B4,0x54321123,0xaaaaaaaa,0xbbbbbbbb,0xcccccccc,0xddddddddd,
0x34667666,0xa46a3aaa,0xe4251e84,0xf31f2Eff,0xfec55267,0x11111111,
0x98765432,0x54376511,0x21323111,0x85433abc,0xcaa44322,0x001234ef
};
IppsBigNumState* M = nBN(sizeof(dataM)/sizeof(dataM[0]),dataM);
// encrypt plain text
IppsBigNumState* C = nBN(sizeof(dataN)/sizeof(dataN[0]));
ippsRSAEncrypt(M, C, RSApub);                       // encryption
IppsBigNumState* Z1 = nBN(sizeof(dataN)/sizeof(dataN[0]));
ippsRSADecrypt(C, Z1, RSAprv);                      // decryption
Ipp32u Rsult;              // compare
ippsCmp_BN(M, Z1, &Rsult); // plain text and decrypted cipher text
cout << Rsult <<endl;      // comparison 0 --> OK
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)Q;
delete [] (Ipp8u*)N;
delete [] (Ipp8u*)E;
delete [] (Ipp8u*)D;
delete [] (Ipp8u*)Pm1;
delete [] (Ipp8u*)Qm1;
delete [] (Ipp8u*)phi;
delete [] (Ipp8u*)C;
delete [] (Ipp8u*)Z1;
delete [] (Ipp8u*)RSAprv;
```

```
delete [] (Ipp8u*)RSApub;
delete [] (Ipp8u*)M;
delete [] (Ipp8u*)ONE;
return 0;}
```

11.3 RSA-OAEP encryption and decryption

RSA-OAEP combines RSA encryption and decryption with OAEP which stands for Optimal Asymmetric Encryption Padding. The details can be found in [PKCS 1].
The IPP library contains general RSA-OAEP schemes for encryption and decryption and additionally the specialized versions corresponding to the hash functions: `MD5, SHA1, SHA224, SHA256, SHA384, SHA512`.
We restrict ourselves to the description and the example in the `SHA1` case. The RSA-OAEP encryption in the SHA1 case is performed by the function `RSAOAEPEncrypt_SHA1`.

```
IppStatus ippsRSAOAEPEncrypt_SHA1(const Ipp8u* Plain, //input:
                                                    //plain text
                        int Plalen,   //input:plain text length
                        const Ipp8u* Label,       //input:label
                        int Lablen,        //input:label length
                        const Ipp8u* Seed,         //input:seed
                        Ipp8u* Ciph,       //output:cipher text
                        IppRSAState* Ctx)   //input:RSA context
```

The corresponding decryption is performed by `RSAOAEPDecrypt_SHA1`.

```
IppStatus ippsRSAOAEPDecrypt_SHA1(const Ipp8u* Ciph,  //input:
                                                   //cipher text
                         const Ipp8u* Label,      //input:label
                         int Lablen,       //input:label length
                         Ipp8u* Decr,   //output:decrypted text
                         int Decrlen,  //input:decr.text length
                         IppRSAState* Ctx)  //input:RSA context
```

```
// RSA-OAEP encryption/decryption
// icpc -xHOST -ipp=crypto   037RSA20.cpp toolb.cpp
#include "ippcore.h"
```

```
#include "ippcp.h"
#include "toolb.h" // nBN, nPRNG, tBN, nRSA
#include<iostream>
int main(){
using namespace std;
static Ipp32u dataE[] = {0x11};
IppsBigNumState* E = nBN(sizeof(dataE)/sizeof(dataE[0]),dataE);
int NbitS=1024;
int PbitS=512;
int Trials=50;
IppsPRNGState* Rand = nPRNG();
Ipp32u privres;
IppsRSAState* RSAprv = nRSA(NbitS,PbitS,IppRSAprivate);
IppStatus w0=
ippsRSAGenerate(E,NbitS,PbitS,Trials,RSAprv, ippsPRNGen, Rand);
cout << "Prv: "<<ippGetStatusString(w0) << endl;
ippsRSAValidate(E, Trials, &privres, RSAprv, ippsPRNGen, Rand);
if(IS_VALID_KEY!=privres)
{  cout <<"invalid priv. keys" <<endl;
   return 0;
}
IppsRSAState* RSApub = nRSA(NbitS,PbitS , IppRSApublic);
IppsBigNumState* N = nBN(NbitS/32);
ippsRSAGetKey(N,IppRSAkeyN,RSAprv);
ippsRSASetKey(N, IppRSAkeyN, RSApub);
ippsRSASetKey(E, IppRSAkeyE, RSApub);
const Ipp8u Plain[] =  "Plain text for RSA-OAEP encr/decryption";
const Ipp8u Seed[] =  "Random text for seeding the hash funct.";
int Plalen=sizeof(Plain);
cout<<"Plain length: "<<Plalen<<endl;
cout<<"Before encr: "<<endl;
for(int k=0;k<Plalen;k++)
cout<<Plain[k];
cout <<endl;
const Ipp8u Label[]="";
int Lablen=sizeof(Label);
Ipp8u* Ciph=new Ipp8u[NbitS/8];
IppStatus w1=ippsRSAOAEPEncrypt_SHA1(Plain,Plalen,Label,Lablen,
```

```
                                                    Seed,Ciph,RSApub);
cout << "Encr: "<<ippGetStatusString(w1) << endl;
int d=Plalen;
Ipp8u DResult[d];
IppStatus w2=ippsRSAOAEPDecrypt_SHA1(Ciph,Label,Lablen,DResult,
                                                          &d,RSAprv);
cout << "Decr: "<<ippGetStatusString(w2) << endl;
cout<<"After decr: "<<endl;
for(int k=0;k<Plalen;k++)
cout<<DResult[k];
cout <<endl;
delete [] (Ipp8u*)E;
delete [] (Ipp8u*)RSAprv;
delete [] (Ipp8u*)RSApub;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)N;
delete [] (Ipp8u*)Ciph;
return 0;}
//Output:
//Prv: ippStsNoErr: No error, it's OK
//Plain length: 39
//Before encr:
//Plain text for RSA-OAEP encryption/decryption
//Encr: ippStsNoErr: No error, it's OK
//Decr: ippStsNoErr: No error, it's OK
//After decr:
//Plain text for RSA-OAEP encryption/decryption
```

11.4 RSA-SSA signature

The IPP library contains general RSA-SSA signature schemes ([PKCS 1]). The specialized versions corresponding to the hash functions: `MD5, SHA1, SHA224, SHA256, SHA384, SHA512` are provided.
As in the previous subsection we restrict ourselves to the description and an example in the `SHA1` case. The RSA-SSA signature in the SHA1 case is created by the function `RSASSASign_SHA1`.

```
IppStatus ippsRSASSASign_SHA1(const Ipp8u* Hash,    //input:
                                                  //hashed text
                      const Ipp8u* Salt,        //input:salt
                      int Salen,         //input:salt length
                      const Ipp8u* Sign,  //output:signature
                      IppRSAState* Ctx)  //input:RSA context
```

The corresponding verification is performed by `RSASSAVerify_SHA1`.

```
IppStatus ippsRSASSAVerify_SHA1(const Ipp8u* Hash,  //input:
                                                  //hashed text
                    const Ipp8u* Sign,      //input:signature
                    IppBool* Valid,//output:verification res.
                    IppRSAState* Ctx)     //input:RSA context
```

```
// RSASSA signature and validation
// icpc -xHOST  -ipp=crypto  037RSA30.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include "toolb.h" // nBN, nPRNG, nRSA
#include<cstring>
#include<iostream>
int main(){
using namespace std;
static Ipp32u dataE[] = {0x11};
IppsBigNumState* E = nBN(sizeof(dataE)/sizeof(dataE[0]),dataE);
int NbitS=1024;
int PbitS=512;
int Trials=50;
IppsPRNGState* Rand = nPRNG();
Ipp32u privres;
IppsRSAState* RSAprv = nRSA(NbitS,PbitS,IppRSAprivate);
IppStatus w0=
ippsRSAGenerate(E,NbitS,PbitS,Trials,RSAprv, ippsPRNGen, Rand);
cout << "Prv: "<<ippGetStatusString(w0) << endl;
ippsRSAValidate(E, Trials, &privres, RSAprv, ippsPRNGen, Rand);
if(IS_VALID_KEY!=privres)
{  cout <<"invalid priv. keys" <<endl;
   return 0;
```

```
}
IppsRSAState* RSApub = nRSA(NbitS,PbitS , IppRSApublic);
IppsBigNumState* N = nBN(NbitS/32);
ippsRSAGetKey(N,IppRSAkeyN,RSAprv);
ippsRSASetKey(N, IppRSAkeyN, RSApub);
ippsRSASetKey(E, IppRSAkeyE, RSApub);
const Ipp8u Mesg[] = "A message sample for RSA-SSA signature.";
int Halen=SHA1_DIGEST_LENGTH/8;
Ipp8u* Hash=new Ipp8u[Halen];
ippsSHA1MessageDigest(Mesg,sizeof(Mesg)-1,Hash);
const Ipp8u Salt[] =  "random text for seeding the sign funct.";
int Salen=sizeof(Salt);
cout<<"Hash length: "<<Halen<<endl;
cout<<"Mesg length: "<<strlen((char*)Mesg)<<endl;
Ipp8u* Sign=new Ipp8u[NbitS/8];
IppStatus wsig=ippsRSASSASign_SHA1(Hash,Salt,Salen,Sign,RSAprv);
cout << "Sign: "<<ippGetStatusString(wsig) << endl;
IppBool Valid;
IppStatus wver=ippsRSASSAVerify_SHA1(Hash,Sign,&Valid,RSApub);
cout << "Ver: "<<ippGetStatusString(wver) << endl;
cout <<"Valid: "<<(IppBool)Valid<<endl;
delete [] (Ipp8u*)E;
delete [] (Ipp8u*)RSAprv;
delete [] (Ipp8u*)RSApub;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)Hash;
delete [] (Ipp8u*)Sign;
delete [] (Ipp8u*)N;
return 0;}
//Output:
//Prv: ippStsNoErr: No error, it's OK
//Hash length: 20
//Mesg length: 39
//Sign: ippStsNoErr: No error, it's OK
//Ver: ippStsNoErr: No error, it's OK
//Valid: 1
```

Chapter 12

Discrete logarithm problem based functions

In this chapter we present the IPP realization of the Digital Signature Algorithm (DSA) and Diffie-Hellman key exchange algorithm (DH). Both are based on the difficulty of the Discrete Logarithm Problem (DLP). A general description of those algorithms can be found in [FIPS 186], [FIPS 186-3] and [PKCS 3].

12.1 Creating a new DLP context

Before the use of any DLP-based function we need an appropriate DLP context preparation, consisting of the following steps.
1. Calculation of DLP context size using `DLPGetSize`.

```
IppStatus ippsDLPGetSize(int PbitS,        //input:P bit-size
                         int QbitS,        //input:Q bit-size
                         int* size)      //output:DLP context
                                              //size in bytes
  //2^(L-1)< P <2^L, 512<=L<=1024, 2^159<Q<2^160, P,Q -primes
                                  //Q is a prime divisor of P-1
```

2. Allocation of the required memory and its initialization with the help of `DLPInit`.

```
IppStatus ippsDLPInit(int PbitS,           //input:P bit-size
                      int QbitS,           //input:Q bit-size
                      IppsDLPState* Ctx)//output:DLP context
```

Using the two mentioned functions one can define an additional function

nDLP, which will be utilized in creating new DLP context in our DLP examples.

```
IppsDLPState* nDLP(int PbitS, int QbitS)
{
   int size;
   ippsDLPGetSize(PbitS, QbitS, &size);
   IppsDLPState *Ctx= (IppsDLPState *)new Ipp8u[ size ];
   ippsDLPInit(PbitS, QbitS, Ctx);
   return Ctx;
}
```

12.2 Setting the DLP parameters

Let P,Q denote the primes satisfying the conditions introduced in the definition of DLPGetSize and let G be a generator of some group of order Q of elements which are invertible modulo P. To set the parameters P, Q, G for the DLP context one can use the DLPSet function.

```
IppStatus ippsDLPSet(const IppsBigNumState* P,//input:P BigNum
                     const IppsBigNumState* Q,  //input:Q BigNum
                     const IppsBigNumState* G,  //input:G BigNum
                     IppsDLPState* Ctx)     //output:DLP context
```

To set a single DLP parameter one can use the DLPSetDP function.

```
IppStatus ippsDLPSetDP(const IppsBigNumState* Par,    //input:
                                    //selected BigNum parameter
                     IppsDLPKeyTag Tag,//input:key specification
                     IppsDLPState* Ctx)     //output:DLP context
 //Tag can be equal to IppDLPkeyP or IppDLPkeyR or IppDLPkeyG
```

Remark. We prefer to use Q-letter instead of R suggested by IppDLPkeyR-tag.

12.3 Retrieving the DLP parameters

To get the parameters P, Q, G of the DLP context one can use the DLPGet function.

```
  IppStatus ippsDLPGet(IppsBigNumState* P,//output:P BigNum
                  IppsBigNumState* Q,     //output:Q BigNum
                  IppsBigNumState* G,     //output:G BigNum
                  IppsDLPState* Ctx)    //input:DLP context
```

To get a single DLP parameter one can use the `DLPGetDP` function.

```
IppStatus ippsDLPSetDP(IppsBigNumState* Par,         //output:
                                    //selected BigNum parameter
                  IppsDLPKeyTag Tag,//input:key specification
                  IppsDLPState* Ctx)      //input:DLP context
 //Tag can be equal to IppDLPkeyP or IppDLPkeyR or IppDLPkeyG
```

12.4 DLP private and public keys

The pair of DLP (private and public) keys can be generated with the help of `DLPGenKeyPair`.

```
 IppStatus ippsDLPGenKeyPair(IppsBigNumState* Prv,  //output:
                                              //private DLP key
                IppsBigNumState* Pub, //output:public DLP key
                IppsDLPState* Ctx,        //input:DLP context
                IppBitSupplier PRNGen,//input:pseudorand.gen.
                void* Rand)    //input:pseudorand gen.context
```

If the private key is given, the public key can be computed using the `DLPPublicKey` function.

```
IppStatus ippsDLPPublicKey(const IppsBigNumState* Prv,//input:
                                                  //private key
                  IppsBigNumState* Pub,   //output:public key
                  IppsDLPState* Ctx)      //input:DLP context
```

The private and public DLP keys can be explicitly set. In this case the function `DLPSetKeyPair` can be used.

```
IppStatus ippsDLPSetKeyPair(const IppsBigNumState* Prv,//input:
                                                    //private key
                const IppsBigNumState* Pub,  //input:public key
                IppsDLPState* Ctx)        //output:DLP context
```

The key pair can be validated with the use of `DLPValidateKeyPair`.

```
IppStatus ippsDLPValidateKeyPair(const IppsBigNumState* Prv,
                                          //input:private key
                const IppsBigNumState* Pub,//input:public key
                IppDLPResult* Reslt, //output:validation res.
                IppsDLPState* Ctx)        //input:DLP context
```

12.5 DLP-DSA parameters generation and validation

To prepare the parameters used in the DSA signature one can utilize the function `DLPGenerateDSA`.

```
IppStatus ippsDLPGenerateDSA(const IppsBigNumState* Seed1,
                                                     //input:seed
                 int Trials,             //input:numb.of tests
                 IppsDLPState* Ctx,       //output:DLP context
                 IppsBigNumState* Seed2,         //output:seed
                 int* Count,                //optional counter
                 IppBitSupplier PRNGen,//input:pseudorand.gen.
                 void* Rand)    //input:pseudorand gen.context
```

The obtained parameters can be validated using `DLPValidateDSA` function.

```
 IppStatus ippsDLPValidateDSA(int Trials,//input:numb.of tests
                 IppDLResult* Result, //output:validation res.
                 IppsDLPState* Ctx,        //input:DLP context
                 IppBitSupplier PRNGen,//input:pseudorand.gen.
                 void* Rand)    //input:pseudorand gen.context
```

12.6 DLP-DSA signature

The DSA signature can be performed using `DLPSignDSA` function.

```
IppStatus ippsDLPSignDSA(const IppsBigNumState* Msg,  //input:
                                                      //message
                  const IppsBigNumState* Priv,//input:priv key
                  IppsBigNumState* SigR,  //output:sign.comp.R
                  IppsBigNumState* SigS,  //output:sign.comp.S
                  IppsDLPState* Ctx)       //input:DLP context
```

In the signature verification procedure one can utilize `DLPVerifyDSA`.

```
IppStatus ippsDLPVerifyDSA(const IppsBigNumState* Msg,//input:
                                                      //message
                  IppsBigNumState* SigR,    //input:sign.comp.R
                  IppsBigNumState* SigS,    //input:sign.comp.S
                  IppDLResult* Result,    //output:verif.result
                  IppsDLPState* Ctx)        //input:DLP context
```

```
// Generation and validation of DLP-DSA parameters
// DSA signature
// icpc -ipp=crypto 038DLP3+.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nPRNG, nBN, nDLP
#include <ctime>
int main(){
using namespace std;
static const int PbitS = 512; // DSA P bit-size
static const int QbitS = 160; // DSA Q order bit-size
srand(time(0));
const int seedBitSize =256;
IppsPRNGState* Rand = nPRNG();
IppsBigNumState* Seed = nBN(seedBitSize/32);
ippsPRNGen_BN(Seed,seedBitSize,Rand);
int nTrials=20;
IppsDLPState* DLPState = nDLP(PbitS, QbitS);
ippsDLPGenerateDSA(Seed,nTrials,DLPState,0,0,ippsPRNGen, Rand);
```

```
                                                    // Generation
IppDLResult result;
ippsDLPValidateDSA( nTrials,&result,DLPState,ippsPRNGen,Rand);
                                                    // Validation
if(result!=ippDLValid)
{cout <<"DSA validation  fail" <<endl;}

Ipp8u message[] = "xyz";
Ipp8u md[SHA1_DIGEST_LENGTH/8];
ippsSHA1MessageDigest(message, sizeof(message)-1, md);
IppsBigNumState* digest = nBN(SHA1_DIGEST_LENGTH/32);
ippsSetOctString_BN(md, SHA1_DIGEST_LENGTH/8, digest);
IppsBigNumState* R = nBN(QbitS/32);
IppsBigNumState* X = nBN(QbitS/32);
IppsBigNumState* Y = nBN(PbitS/32);
IppsBigNumState* S = nBN(QbitS/32);
ippsDLPGenKeyPair(X, Y, DLPState, ippsPRNGen, Rand);
                                        // Key pair generation
ippsDLPSetKeyPair(X, Y, DLPState);
IppDLResult result1;
ippsDLPValidateKeyPair(X, Y, &result1,DLPState);
                                        // Key pair validation
if(result1!=ippDLValid)
   {cout <<"Key validation  fail" <<endl;}
ippsDLPSignDSA(digest,X,R,S,DLPState);  // DSA signature
IppDLResult result2;
ippsDLPVerifyDSA(digest,R,S,&result2, DLPState);  //Sign.verif.
if(result2!=ippDLValid)
   {cout <<"Signature validation  fail" <<endl;}
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)Seed;
delete [] (Ipp8u*)DLPState;
delete [] (Ipp8u*)digest;
delete [] (Ipp8u*)R;
delete [] (Ipp8u*)X;
delete [] (Ipp8u*)Y;
delete [] (Ipp8u*)S;
return 0;}
```

```
// No output --> OK
```

In the next example the parameters are manually set.

```
// DLP primitives with given P,Q,G, DSA signature
// icpc -ipp=crypto 038DLP1+.cpp toolb.cpp
#include "ippcp.h"
#include<iostream>
#include "toolb.h" // nBN, nDLP, nPRNG
int main(){
using namespace std;
static const int PbitS = 512; // DSA parameter P bit-size
static const int QbitS = 160; // DSA order parameter Q bit-size
Ipp32u Pdata[]=
{0x47e8eabd,0xb265c6b6,0x545d54c5,0x69cc3d19,0x99df7412,
0x1bd78998,0x20cd4fce,0x64aabfd0,0x3a0c55a6,0x1c61954c,
0x1ace1d60,0x3418cda9,0xe45f6aff,0x52f29367,0x7b95c4a9,
0xb097e8b6};
Ipp32u Qdata[]=
{0xa7824e0f,0x03eda0c2,0xffc495da,0x2732b2a5,0xe436207e};
Ipp32u Gdata[]=
{0xfb687d23,0x0c622a5d,0x8a11421a,0xd9a36e84,0x9ba7073c,
0x24c6db1a,0x72d20cd5,0x4842cd31,0x0b899903,0xff108a78,
0xf24ebda6,0x571b94ee,0x7ed06f28,0x51559df6,0x774e7fbe,
0x5d4ff9a8};
IppsBigNumState* P = nBN(PbitS/32,Pdata);
IppsBigNumState* Q = nBN(QbitS/32,Qdata);
IppsBigNumState* G = nBN(PbitS/32,Gdata);
IppsDLPState* DLPState = nDLP(PbitS, QbitS);
ippsDLPSet(P, Q, G, DLPState);
Ipp32u Xdata[QbitS/32]={0x0};
Ipp32u Ydata[PbitS/32]={0x0};
IppsBigNumState* X = nBN(QbitS/32);
IppsBigNumState* Y = nBN(PbitS/32);
IppsPRNGState* pRand = nPRNG();
ippsDLPGenKeyPair(X, Y, DLPState, ippsPRNGen, pRand);
                                  // Key pair generation
ippsDLPSetKeyPair(X, Y, DLPState);
IppDLResult result;
```

```
ippsDLPValidateKeyPair(X, Y, &result,DLPState);
                                    // Key pair validation
if(result!=ippDLValid)
   {cout <<"validation 1 fail" <<endl;}
Ipp8u message[] = "xyz";
Ipp8u md[SHA1_DIGEST_LENGTH/8];
ippsSHA1MessageDigest(message, sizeof(message)-1, md);
IppsBigNumState* digest = nBN(SHA1_DIGEST_LENGTH/32);
ippsSetOctString_BN(md, SHA1_DIGEST_LENGTH/8, digest);
IppsBigNumState* R = nBN(QbitS/32,Xdata);
IppsBigNumState* S = nBN(QbitS/32,Xdata);
ippsDLPSignDSA(digest,X,R,S,DLPState);          //DSA signature
IppDLResult result1;
ippsDLPVerifyDSA(digest,R,S,&result1, DLPState);//Sign.verif.
if(result1!=ippDLValid)
   {cout <<"validation 2 fail" <<endl;}
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)Q;
delete [] (Ipp8u*)G;
delete [] (Ipp8u*)DLPState;
delete [] (Ipp8u*)pRand;
delete [] (Ipp8u*)digest;
delete [] (Ipp8u*)X;
delete [] (Ipp8u*)Y;
delete [] (Ipp8u*)R;
delete [] (Ipp8u*)S;
return 0;}
// No output  --> OK
// Repeat in case of failure
```

12.7 Diffie-Hellman key exchange

The DLP based functions allow for defining some secret BigNum shared by users A, B. It is equal to $\texttt{PubB}^{\texttt{PrvA}}$ (mod P) for the user A and $\texttt{PubA}^{\texttt{PrvB}}$ (mod P) for the user B. Since $\texttt{PubA} = \texttt{G}^{\texttt{PrvA}}$ (mod P) and $\texttt{PubB} = \texttt{G}^{\texttt{PrvB}}$ (mod P), both calculations give the same result. Before the shared secret is defined, it is necessary to generate appropriate domain parameters using `DLPGenerateDH` function.

```
IppStatus ippsDLPGenerateDH(const IppsBigNumState* Seed1,
                                                  //input:seed
                int Trials,             //input:numb.of tests
                IppsDLPState* Ctx,       //output:DLP context
                IppsBigNumState* Seed2,         //output:seed
                int* Count,                //optional counter
                IppBitSupplier PRNGen,//input:pseudorand.gen.
                void* Rand)    //input:pseudorand gen.context
```

The obtained parameters can be validated with the help of `DLPValidateDH`.

```
IppStatus ippsDLPValidateDH(int Trials, //input:numb.of tests
                IppDLResult* Result, //output:validation res.
                IppsDLPState* Ctx,        //input:DLP context
                IppBitSupplier PRNGen,//input:pseudorand.gen.
                void* Rand)    //input:pseudorand gen.context
```

The shared secret can be computed using the function `DLPSharedSecretDH`.

```
IppStatus ippsDLPSharedSecretDH(const IppsBigNumState* Prv,
                                             //input:priv.key
            const IppsBigNumState* Pub,        //input:pub.key
            const IppsBigNumState* Share,//output:shared.secr.
            IppsDLPState* Ctx,             //input:DLP context
```

```
// Generation and validation of DLP-DH parameters
// icpc -ipp=crypto 038DLP4+.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nPRNG, nBN, nDLP
#include <ctime>

int main(){
using namespace std;
static const int PbitS = 512; // DH P bit-size
static const int QbitS = 160; // DH Q order bit-size
srand(time(0));
```

```
const int seedBitSize =256;
IppsPRNGState* Rand = nPRNG();
IppsBigNumState* Seed = nBN(seedBitSize/32);
ippsPRNGen_BN(Seed,seedBitSize,Rand);
int nTrials=20;
IppsDLPState* DLPState = nDLP(PbitS, QbitS);
ippsDLPGenerateDH(Seed,nTrials,DLPState,0,0,ippsPRNGen,Rand);
                                                    // Generation
IppDLResult result;
ippsDLPValidateDH(nTrials,&result,DLPState,ippsPRNGen,Rand);
                                                    // Validation
if(result!=ippDLValid)
{cout <<"DH validation  fail" <<endl;}
//cout <<"DHValidation: "<<ippsDLGetResultString(result)<<endl;
IppsBigNumState* PrvA = nBN(QbitS/32);
IppsBigNumState* PrvB = nBN(QbitS/32);
IppsBigNumState* PubA = nBN(PbitS/32);
IppsBigNumState* PubB = nBN(PbitS/32);
IppsBigNumState* ShareA = nBN(PbitS/32);
IppsBigNumState* ShareB = nBN(PbitS/32);
ippsDLPGenKeyPair(PrvA, PubA, DLPState, ippsPRNGen, Rand);
                                        //A Key pair generation
ippsDLPSetKeyPair(PrvA, PubA, DLPState);
IppDLResult result1;
ippsDLPValidateKeyPair(PrvA, PubA, &result1,DLPState);
                                        //A Key pair validation
if(result1!=ippDLValid)
   {cout <<"A Key validation  fail" <<endl;}
ippsDLPGenKeyPair(PrvB, PubB, DLPState, ippsPRNGen, Rand);
                                        //B Key pair generation
ippsDLPSetKeyPair(PrvB, PubB, DLPState);
IppDLResult result2;
ippsDLPValidateKeyPair(PrvB, PubB, &result2,DLPState);
                                        //B Key pair validation
if(result2!=ippDLValid)
   {cout <<"B Key validation  fail" <<endl;}
ippsDLPSharedSecretDH(PrvA,PubB,ShareA,DLPState);
ippsDLPSharedSecretDH(PrvB,PubA,ShareB,DLPState);
```

```
Ipp32u result3;
ippsCmp_BN(ShareA,ShareB,&result3);//Check if Shared keys agree;
                                   // 0 --> OK
cout<<"Shared Secr. comparison: "<<result3<<endl;
Ipp32u result4;
ippsCmp_BN(PrvA,PrvB,&result4); // Check if Priv keys differ;
                                // no 0 --> OK
cout<<"Priv Key comparison: "<<result4<<endl;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)Seed;
delete [] (Ipp8u*)DLPState;
delete [] (Ipp8u*)PrvA;
delete [] (Ipp8u*)PrvB;
delete [] (Ipp8u*)PubA;
delete [] (Ipp8u*)PubB;
delete [] (Ipp8u*)ShareA;
delete [] (Ipp8u*)ShareB;
return 0;}

//Output:
//Shared Secr. comparison: 0
//Priv Key comparison: 1
```

Chapter 13

Elliptic curve cryptography over prime finite field

A short introduction to elliptic curves can be found in [SMART]. A detailed, freely available description of elliptic curve cryptography is contained in [SEC 1], [SEC 2].

13.1 Creating a new ECCP context

Before we start to use the elliptic curves we need to prepare an appropriate context. A new ECCP context can be created with the help of `ECCPGetSize` and `ECCPInit`.

```
IppStatus ippsECCPGetSize(int bitS,    //input:curve bit-size
                          int* size)    //output:ECCP context
                                              //size in bytes
```

The size obtained from `ECCPGetSize` can be used to allocate a sufficient amount of memory for the ECCP context. The memory can be initialized with the use of `ECCPInit`.

```
IppStatus ippsECCPInit(int bitS,       //input:curve bit-size
                    IppsECCPState* ECCP)//output:ECCP context
```

In examples we shall use the function `nECCP` which performs the just mentioned steps.

```
IppsECCPState* nECCP(int bitS)
{ int size;
```

```
  ippsECCPGetSize(bitS,&size);
  IppsECCPState* ECCP=(IppsECCPState *)(new Ipp8u [size]);
  ippsECCPInit(bitS,ECCP);
  return ECCP;
}
```

13.2 Standard elliptic curve setup

The shortest way to set the parameters for the ECCP context is to use the function `ECCPSetStd`.

```
IppStatus ippsECCPSetStd(IppECCType flag,  //input:curve flag
                    IppsECCPState* ECCP)//output:ECCP context
```

The `flag` can take one of the following values:

```
IppECCPStd112r1,    IppECCPStd112r2,    IppECCPStd128r1,
IppECCPStd128r2,    IppECCPStd160r1,    IppECCPStd160r2,
IppECCPStd192r1,    IppECCPStd224r1,    IppECCPStd256r1,
IppECCPStd384r1,    IppECCPStd521r1.
```

The parameters of the elliptic curve can be verified using the function `ECCPValidate`.

```
IppStatus ippsECCPValidate( int Trials,  //input:numb.of tests
                  IppECResult* Result, //output:validation res.
                  IppsECCPState* Ctx,      //input:ECCP context
                  IppBitSupplier PRNGen,//input:pseudorand.gen.
                  void* Rand)    //input:pseudorand gen.context
```

All the introduced functions are used in the following simple example.

```
// Definition and validation of the standard EC
//icpc -xHOST -ipp=crypto   040ECCP00.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
```

```
IppsECCPState* ECCP1=nECCP(160);
ippsECCPSetStd(IppECCPStd160r1,ECCP1);
IppsPRNGState* Rand=nPRNG();
int nTrials=20;
IppECResult Result;
ippsECCPValidate(nTrials,&Result,ECCP1,ippsPRNGen,Rand);
cout << "EC Validation: "<<ippsECCGetResultString(Result)<<endl;
delete [] (Ipp8u*)ECCP1;
delete [] (Ipp8u*)Rand;
return 0;
}
//Output:
//EC Validation: Validation pass successfully
```

13.3 Elliptic curve parameters

To define an elliptic curve $\mathtt{E_p}$:

$$\mathtt{y}^2 = \mathtt{x}^3 + \mathtt{Ax} + \mathtt{B} \pmod{\mathtt{P}}$$

(over finite field `GF(P)`) it is necessary to specify the `BigNum` parameters `P, A, B`. We also need to define the `BigNum` x, y coordinates `GX, GY` of the base point `G` (i.e. the point generating some subgroup of points on the curve). It is also necessary to set the `BigNum` - the `Order` of the group generated by `G` and the cofactor `Cof` (i.e. the quotient $\#\mathtt{E_p}/\mathtt{Order}$, where $\#\mathtt{E_p}$ denotes the number of all points on $\mathtt{E_p}$). All the parameters can be set with the help of `ECCPSet`.

```
IppStatus ippsECCPSet(const IppsBigNumState* P,//input:modulus
          const IppsBigNumState* A,          //input:coeff.A
          const IppsBigNumState* B,          //input:coeff.B
          const IppsBigNumState* GX,//input:X-coordinate of G
          const IppsBigNumState* GY,//input:Y-coordinate of G
          const IppsBigNumState* Ord,     //input:Group order
          int Cof,                           //input:cofactor
          IppsECCPState* Ctx)           //output:ECCP context
```

```
//Setting EC parameters and EC validation
//icpc -xHOST  -ipp=crypto  040ECCP02.cpp toolb.cpp
```

```
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=224;
IppsECCPState* ECC=nECCP(bitS);
const Ipp32u P[]= {0x7EC8C0FF,0x97DA89F5,0xB09F0757,
        0x75D1D787,0x2A183025,0x26436686,0xD7C134AA};
const Ipp32u A[]= {0xCAD29F43,0xB0042A59,0x4E182AD8,
        0xC1530B51,0x299803A6,0xA9CE6C1C,0x68A5E62C};
const Ipp32u B[]= {0x386C400B,0x66DBB372,0x3E2135D2,
        0xA92369E3,0x870713B1,0xCFE44138,0x2580F63C};
const Ipp32u GX[]={0xEE12C07D,0x4C1E6EFD,0x9E4CE317,
        0xA87DC68C,0x340823B2,0x2C7E5CF4,0x0D9029AD};
const Ipp32u GY[]={0x761402CD,0xCAA3F6D3,0x354B9E99,
        0x4ECDAC24,0x24C6B89E,0x72C0726F,0x58AA56F7};
const Ipp32u Or[]={0xa5a7939f,0x6ddebca3,0xd116bc4b,
       0x75d0fb98,0x2a183025,0x26436686,0xd7c134aa};
IppsBigNumState* pP = nBN(sizeof(P)/4,P);
IppsBigNumState* pA = nBN(sizeof(A)/4,A);
IppsBigNumState* pB = nBN(sizeof(B)/4,B);
IppsBigNumState* pGX = nBN(sizeof(GX)/4,GX);
IppsBigNumState* pGY = nBN(sizeof(GY)/4,GY);
IppsBigNumState* pOr = nBN(sizeof(Or)/4,Or);
int cof=0x1;
ippsECCPSet(pP,pA,pB,pGX,pGY,pOr,cof,ECC);
IppsPRNGState* Rand=nPRNG();
int Trials=20;
IppECResult Result;
ippsECCPValidate(Trials,&Result,ECC,ippsPRNGen,Rand);
cout<<"EC Validation: "<<ippsECCGetResultString(Result)<<endl;
delete [] (Ipp8u*)pP;
delete [] (Ipp8u*)pA;
delete [] (Ipp8u*)pB;
delete [] (Ipp8u*)pGX;
delete [] (Ipp8u*)pGY;
```

```
delete [] (Ipp8u*)pOr;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)ECC;
return 0;
}
//Output:
//EC Validation: Validation pass successfully
```

If the elliptic curve is already defined then all its parameters can be retrieved from the context with the use of `ECCPGet`.

```
IppStatus ippsECCPGet(IppsBigNumState* P,    //output:modulus
               IppsBigNumState* A,           //output:coeff.A
               IppsBigNumState* B,           //output:coeff.B
               IppsBigNumState* GX,//output:X-coordinate of G
               IppsBigNumState* GY,//output:Y-coordinate of G
               IppsBigNumState* Ord,     //output:Group order
               int Cof,                     //output:cofactor
               IppsECCPState* Ctx)       //input:ECCP context
```

```
 IppStatus ippsECCPGetOrderBitSize(int bitS, //output:curve
                                                 //bit-size
                   IppsECCPState* ECC)//input:ECCP context
```

```
// Retrieving parameters of the standard EC
//icpc -xHOST -ipp=crypto   040ECCP01.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=160;
IppsECCPState* ECCP=nECCP(bitS);
ippsECCPSetStd(IppECCPStd160r1,ECCP);
IppsBigNumState* P = nBN(bitS/32); //Modulus P (Prime Number)
IppsBigNumState* A = nBN(bitS/32);  //Coeff A
IppsBigNumState* B = nBN(bitS/32);  //Coeff B
IppsBigNumState* GX = nBN(bitS/32); //Base Point Coord. X
IppsBigNumState* GY = nBN(bitS/32); //Base Point Coord. Y
```

```
int bitSize;
ippsECCPGetOrderBitSize(&bitSize,ECCP);
IppsBigNumState* Order=nBN(bitSize/32+1);//BasePoint Order
int  cofactor;                          //cofactor=#E/Order
ippsECCPGet(P,A,B,GX,GY,Order,&cofactor,ECCP);//Get EC param.
tBN("P:\n",P); // P:fffffffffffffffffffffffffffffff7fffffff
tBN("A:\n",A); // A:fffffffffffffffffffffffffffffff7ffffffc
tBN("B:\n",B); // B:1c97befc54bd7a8b65acf89f81d4d4adc565fa45
tBN("Order:\n",Order);
          //Order:10000000000000000001f4c8f927aed3ca752257
cout<<"Cof: "<<cofactor<<endl;       //Cof: 1
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)A;
delete [] (Ipp8u*)B;
delete [] (Ipp8u*)GX;
delete [] (Ipp8u*)GY;
delete [] (Ipp8u*)Order;
delete [] (Ipp8u*)ECCP;
return 0;
}
```

13.4 Points on the elliptic curve

The points on an elliptic curve have separate context which must be prepared and initialized before any operation is performed. First we have to compute an appropriate size of memory. This evaluation can be done with the help of `ECCPPointGetSize`.

```
IppStatus ippsECCPPointGetSize(int bitS,//input:curve bit-size
                               int* Psize) //output:EC point size
```

As a next step, an appropriate amount of memory must be allocated and initialized.

```
IppStatus ippsECCPPointInit(int bitS,   //input:curve bit-size
                    IppsECCPPointState* P)//output:initialized
                                         //EC point context
```

In examples we shall define new EC points contexts using the function

nECCPPoint.

```
IppsECCPPointState* nECCPPoint(int bitS,int Psize)
{
IppsECCPPointState* Point=(IppsECCPPointState *)
                                              (new Ipp8u [Psize]);
ippsECCPPointInit(bitS,Point);
return Point;
}
```

The coordinates of the point can be set with the use of ECCPSetPoint.

```
 IppStatus ippsECCPSetPoint(const IppsBigNumState* X,//input:
                                                    X-coordinate
               const IppsBigNumState* Y, //input:Y-coordinate
               IppsECCPPointState* P,       //output:EC point
               IppsECCPState* ECCP)      //input:ECCP context
```

There is a special function defining the point at infinity.

```
IppStatus ippsECCPSetPointAtInfinity(IppsECCPPointState* O,
                                  //output:point at infinity
                     IppsECCPState* ECCP)//input:ECCP context
```

The point can be validated using the function ECCPCheckPoint.

```
 IppStatus ippsECCPCheckPoint(const IppsECCPPointState* P,
                                              //input:EC point
               IppECResult* Result,//output:validation result
               IppsECCPState* ECCP)      //input:ECCP context
```

The x, y coordinates can be retrieved with the use of ECCPGetPoint.

```
 IppStatus ippsECCPGetPoint(IppsBigNumState* X,    //output:
                                                  X-coordinate
                   IppsBigNumState* Y,//output:Y-coordinate
                   IppsECCPPointState* P,  //input:EC point
                   IppsECCPState* ECCP)//input:ECCP context
```

```
// Definition and validation of the  point on EC
//icpc -xHOST -ipp=crypto   040ECCP03.cpp toolb.cpp
```

```
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=160;
IppsECCPState* ECCP=nECCP(bitS);
ippsECCPSetStd(IppECCPStd160r1,ECCP);
int Psize;
ippsECCPPointGetSize(bitS,&Psize);
IppsECCPPointState* P=nECCPPoint(bitS,Psize);
const Ipp32u GX[]=
{0x13cbfc82,0x68c38bb9,0x46646989,0x8ef57328,0x4a96b568};
const Ipp32u GY[]=
{0x7ac5fb32,0x04235137,0x59dcc912,0x3168947d,0x23a62855};
IppsBigNumState* X = nBN(bitS/32,GX);
IppsBigNumState* Y = nBN(bitS/32,GY);
ippsECCPSetPoint(X,Y,P,ECCP);
IppsBigNumState* X1 = nBN(bitS/32,GX);
IppsBigNumState* Y1 = nBN(bitS/32,GY);
ippsECCPGetPoint(X1,Y1,P,ECCP);
tBN("X: ",X1);
IppECResult Result;
ippsECCPCheckPoint(P,&Result,ECCP);
cout<<"CheckPoint: "<<ippsECCGetResultString(Result)<<endl;
delete [] (Ipp8u*)X;
delete [] (Ipp8u*)Y;
delete [] (Ipp8u*)X1;
delete [] (Ipp8u*)Y1;
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)ECCP;
return 0;
}
//Output:
//X: 4a96b5688ef573284664698968c38bb913cbfc82
//CheckPoint: Validation pass successfully
```

13.5 Arithmetic of elliptic curves

In IPP we have at our disposal the operations of comparison, negation, addition and scalar multiplication of points on an elliptic curve. First we describe the syntax of the corresponding functions.

1. The comparison of two points on an elliptic curve can be performed using `ECCPComparePoint`.

```
IppStatus ippsECCPComparePoint(const IppsECCPPointState* P,
                                                //input:EC point
            const IppsECCPPointState* Q,  //input:EC point
            IppECResult* Result,//output:comparison result
            IppsECCPState* ECCP)      //input:ECCP context
```

2. The negation (-P) of a point on an elliptic curve one can obtain with the help of `ECCPNegativePoint`.

```
IppStatus ippsECCPNegativePoint(const IppsECCPPointState* P,
                                                 //input:EC point
                  IppsECCPPointState* Q,//output:EC point -P
                  IppsECCPState* ECCP)  //input:ECCP context
```

3. The addition of two points on an elliptic curve can be performed by the function `ECCPAddPoint`.

```
IppStatus ippsECCPAddPoint(const IppsECCPPointState* P,
                                                //input:EC point
              const IppsECCPPointState* Q, //input:EC point
              IppsECCPPointState* R,//output:EC point R=P+Q
              IppsECCPState* ECCP)     //input:ECCP context
```

4. The multiplication of a point on an elliptic curve by a `BigNum` can be accomplished by `ECCPMulPointScalar`.

```
IppStatus ippsECCPMulPointScalar(const IppsECCPPointState* P,
                                                //input:EC point
           const IppsBigNumState* M,//input:BigNum multiplier
           IppsECCPPointState* R,     //output:EC point R=M*P
           IppsECCPState* ECCP)          //input:ECCP context
```

```
// EC arithmetic
//icpc -xHOST -ipp=crypto   040ECCP04.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=160;
IppsECCPState* ECCP=nECCP(bitS);
ippsECCPSetStd(IppECCPStd160r1,ECCP);
int Psize;
ippsECCPPointGetSize(bitS,&Psize);
IppsECCPPointState* P=nECCPPoint(bitS,Psize);
const Ipp32u GX[]=
{0x13cbfc82,0x68c38bb9,0x46646989,0x8ef57328,0x4a96b568};
const Ipp32u GY[]=
{0x7ac5fb32,0x04235137,0x59dcc912,0x3168947d,0x23a62855};
IppsBigNumState* X = nBN(bitS/32,GX);
IppsBigNumState* Y = nBN(bitS/32,GY);
ippsECCPSetPoint(X,Y,P,ECCP);
IppsECCPPointState* R=nECCPPoint(bitS,Psize);
ippsECCPNegativePoint(P,R,ECCP);                  // -P=R
IppsECCPPointState* Q=nECCPPoint(bitS,Psize);
ippsECCPAddPoint(P,R,Q,ECCP);                     // P+(-P)=Q
IppsECCPPointState* O=nECCPPoint(bitS,Psize);
ippsECCPSetPointAtInfinity(O,ECCP);
IppECResult Result;
ippsECCPComparePoint(Q,O,&Result,ECCP);           // P+(-P)==O
cout << "P+(-P)==O: "<<ippsECCGetResultString(Result)<<endl;
int bitSize;
ippsECCPGetOrderBitSize(&bitSize,ECCP);
IppsBigNumState* Ord = nBN(bitSize/32+1);
IppsBigNumState* X1 = nBN(bitS/32);
IppsBigNumState* Y1 = nBN(bitS/32);
int cof;
ippsECCPGet(X1,X1,Y1,X1,Y1,Ord,&cof,ECCP);
ippsECCPMulPointScalar(P,Ord,R,ECCP);             // Ord*P=R
ippsECCPComparePoint(O,R,&Result,ECCP);           // Ord*P==O
```

```
cout << "P^Ord==0: "<<ippsECCGetResultString(Result)<<endl;
delete [] (Ipp8u*)X;
delete [] (Ipp8u*)Y;
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)R;
delete [] (Ipp8u*)Q;
delete [] (Ipp8u*)O;
delete [] (Ipp8u*)Ord;
delete [] (Ipp8u*)X1;
delete [] (Ipp8u*)Y1;
delete [] (Ipp8u*)ECCP;
return 0;
}
//Output:
//P+(-P)==O: Points are equal
//Ord*P==O: Points are equal
```

13.6 ECCP cryptosystem keys

To use the ECCP cryptographic primitives we have to prepare private and public keys.

- The private key `Prv` is a `BigNum` in the interval `[1,Ord-1]`, where `Ord` is the order of the base point `G` (generating the group of points under consideration). The bit size of `Prv` must be less than the parameter retrieved by the function `ECCPGetOrderSize`.
- The public key `Pub` is the `ECCPPoint` of the form `Prv*G`.

The keys can be generated with the help of `ECCPGenKeyPair`.

```
IppStatus ippsECCPGenKeyPair( IppsBigNumState* Prv, //output:
                                                    //private key
                IppsECCPPointState* Pub,  //output:public key
                IppsECCPState* Ctx,      //input:ECCP context
                IppBitSupplier PRNGen,//input:pseudorand.gen.
                void* Rand)    //input:pseudorand gen.context
```

As we have noticed, the public key can be computed from the given private key. In computations one can use the function `ECCPPublicKey`.

```
IppStatus ippsECCPPublicKey( const IppsBigNumState* Prv,
                                          //input:private key
                 IppsECCPPointState* Pub,//output:public key
                 IppsECCPState* Ctx)    //input:ECCP context
```

The obtained keys can be validated with the help of `ECCPValidateKeyPair`.

```
IppStatus ippsECCPValidateKeyPair(const IppsBigNumState* Prv,
                                           //input:private key
             const IppsECCPPointState* Pub,//input:public key
             IppECResult* Reslt,   //output:validation result
             IppsECCPState* Ctx)         //input:ECCP context
```

The keys can be set with the use of the function `ECCPSetKeyPair`.

```
IppStatus ippsECCPSetKeyPair(const IppsBigNumState* Prv,
                                           //input:private key
             const IppsECCPPointState* Pub,//input:public key
             IppBool KeyType,                //input:key type
             IppsECCPState* Ctx)         //input:ECCP context
//KeyType values:  ippTrue (regular) or ippFalse (ephemeral)
```

```
// ECCP Keys
// icpc -ipp=crypto 040ECCP05.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, nPRNG, nECCP, nECCPPoint
#include<ctime>

int main(){
using namespace std;
srand(time(0));
int  size;
int bitS=256;
// Standard EC 256r1 Setting
ippsECCPGetSize(bitS,&size);
IppsECCPState* ECCP=nECCP(bitS);
ippsECCPSetStd(IppECCPStd256r1,ECCP);
```

```
// Regular Keys
int PointSize;
ippsECCPPointGetSize(bitS,&PointSize);
IppsBigNumState* Priv = nBN(bitS/32);
IppsECCPPointState* Pub=nECCPPoint(bitS,PointSize);
IppsPRNGState* Rand=nPRNG();
ippsECCPGenKeyPair(Priv,Pub,ECCP,ippsPRNGen,Rand);
ippsECCPSetKeyPair(Priv,Pub,ippTrue,ECCP);
IppECResult Result;
ippsECCPValidateKeyPair(Priv,Pub,&Result,ECCP);
cout << "regKeyPairValidation: "<< ippsECCGetResultString(Result)
                                                        << endl;
// Output: regKeyPairValidation: Validation pass successfully
// Ephemeral Keys
IppsBigNumState* ephPriv = nBN(bitS/32);
IppsECCPPointState* ephPub=nECCPPoint(bitS,PointSize);
ippsECCPGenKeyPair(ephPriv,ephPub,ECCP,ippsPRNGen,Rand);
ippsECCPSetKeyPair(ephPriv,ephPub,ippFalse,ECCP);
ippsECCPPublicKey(ephPriv,ephPub,ECCP);
ippsECCPValidateKeyPair(ephPriv,ephPub,&Result,ECCP);
cout << "ephKeyPairValidation: "<< ippsECCGetResultString(Result)
                                                        << endl;
// Output: ephKeyPairValidation: Validation pass successfully
delete [] (Ipp8u*)ephPriv;
delete [] (Ipp8u*)ephPub;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)Pub;
delete [] (Ipp8u*)Priv;
delete [] (Ipp8u*)ECCP;
return 0;
}
```

13.7 ECCP based digital signature

A digital signature of a message consists of two `BigNums`, let's say `SignX` and `SignY`. To compute the signature, we first have to apply a hash function to the message. As a result we obtain the message digest. The message digest and user's private key allow for creating user's signature. The appropriate

procedure can be accomplished with the help of `ECCPSignDSA`.

```
 IppStatus ippsECCPSignDSA(const IppsBigNumState* Digest,
                                          //input:message digest
              const IppsBigNumState * Prv, //input:private key
              IppsBigNumState * SignX,//output:X comp.of sign.
              IppsBigNumState * SignY,//output:Y comp.of sign.
              IppsECCPState* Ctx)         //input:ECCP context
```

The message digest and user's regular public key are used in the verification procedure which is performed by the function `ECCPVerifyDSA`.

```
IppStatus ippsECCPVerifyDSA(const IppsBigNumState* Digest,
                                          //input:message digest
          const IppsBigNumState * SignX,//input:X comp.of sign.
          const IppsBigNumState * SignY,//input:Y comp.of sign.
          IppsECCPState* Ctx)              //input:ECCP context
```

There are also Nyberg-Rueppel versions of the signature scheme.

```
 IppStatus ippsECCPSignNR(const IppsBigNumState* Digest,
                                          //input:message digest
              const IppsBigNumState * Prv, //input:private key
              IppsBigNumState * SignX,//output:X comp.of sign.
              IppsBigNumState * SignY,//output:Y comp.of sign.
              IppsECCPState* Ctx)         //input:ECCP context
```

```
IppStatus ippsECCPVerifyNR(const IppsBigNumState* Digest,
                                          //input:message digest
          const IppsBigNumState * SignX,//input:X comp.of sign.
          const IppsBigNumState * SignY,//input:Y comp.of sign.
          IppsECCPState* Ctx)              //input:ECCP context
```

```
// ECCP DSA signature
// icpc -ipp=crypto 040ECCP06.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, nPRNG, nECCP, nECCPPoint
```

```
#include<ctime>

int main(){
using namespace std;
srand(time(0));
int  size;
int bitS=256;
// Standard EC 256r1 setting
ippsECCPGetSize(bitS,&size);
IppsECCPState* ECCP=nECCP(bitS);
ippsECCPSetStd(IppECCPStd256r1,ECCP);
// Regular Keys
int PointSize;
ippsECCPPointGetSize(bitS,&PointSize);
IppsBigNumState* Priv = nBN(bitS/32);
IppsECCPPointState* Pub=nECCPPoint(bitS,PointSize);
IppsPRNGState* Rand=nPRNG();
ippsECCPGenKeyPair(Priv,Pub,ECCP,ippsPRNGen,Rand);
ippsECCPSetKeyPair(Priv,Pub,ippTrue,ECCP);
IppECResult Result;
ippsECCPValidateKeyPair(Priv,Pub,&Result,ECCP);
cout<<"regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    <<endl;
// Output: regKeyPairValidation: Validation pass successfully
// Ephemeral Keys
IppsBigNumState* ephPriv = nBN(bitS/32);
IppsECCPPointState* ephPub=nECCPPoint(bitS,PointSize);
ippsECCPGenKeyPair(ephPriv,ephPub,ECCP,ippsPRNGen,Rand);
ippsECCPSetKeyPair(ephPriv,ephPub,ippFalse,ECCP);
ippsECCPPublicKey(ephPriv,ephPub,ECCP);
ippsECCPValidateKeyPair(ephPriv,ephPub,&Result,ECCP);
cout<<"ephKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    <<endl;
// Output: ephKeyPairValidation: Validation pass successfully

// Digest
Ipp32u ddat[] = {
      0xBEAD208B, 0x5E668076, 0x2ABF62E3, 0xDB7C000};
```

```
IppsBigNumState* digest=nBN(bitS/32,ddat);

// Signature
IppsBigNumState* SignX = nBN(bitS/32);
IppsBigNumState* SignY = nBN(bitS/32);
IppStatus w;
w=ippsECCPSignDSA(digest,Priv,SignX,SignY,ECCP);
//w=ippsECCPSignNR(digest,Priv,SignX,SignY,ECCP);
cout << "Sign: "<<ippGetStatusString(w) << endl;
// Output: Sign: ippStsNoErr: No error, it's OK

//Verification
w=ippsECCPVerifyDSA(digest,SignX,SignY,&Result,ECCP);
//w=ippsECCPVerifyNR(digest,SignX,SignY,&Result,ECCP);
cout<<"VerifyDSA: "<<ippsECCGetResultString(Result)<<endl;
// Output: VerifyDSA:  Validation pass successfully
cout <<"Verify: "<< ippGetStatusString(w) << endl;
// Output: Verify: ippStsNoErr: No error, it's OK
delete [] (Ipp8u*)SignX;
delete [] (Ipp8u*)SignY;
delete [] (Ipp8u*)digest;
delete [] (Ipp8u*)ephPub;
delete [] (Ipp8u*)ephPriv;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)Pub;
delete [] (Ipp8u*)Priv;
delete [] (Ipp8u*)ECCP;
return 0;
}
```

In the next example the elliptic curve parameters and the regular keys are explicitly defined.

```
// ECDSA with explicitly defined parameters
//icpc -xHOST  -ipp=crypto  040ECCP11.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
```

```
int main(){
using namespace std;
int  size;
int bitS=224;
ippsECCPGetSize(bitS,&size);
IppsECCPState* pECC=nECCP(bitS);
const Ipp32u P[]={0x7EC8C0FF,0x97DA89F5,0xB09F0757,0x75D1D787,
                  0x2A183025,0x26436686,0xD7C134AA};
const Ipp32u A[]= {0xCAD29F43,0xB0042A59,0x4E182AD8,0xC1530B51,
                   0x299803A6,0xA9CE6C1C,0x68A5E62C};
const Ipp32u B[]= {0x386C400B,0x66DBB372,0x3E2135D2,0xA92369E3,
                   0x870713B1,0xCFE44138,0x2580F63C};
const Ipp32u GX[]={0xEE12C07D,0x4C1E6EFD,0x9E4CE317,0xA87DC68C,
                   0x340823B2,0x2C7E5CF4,0x0D9029AD};
const Ipp32u GY[]={0x761402CD,0xCAA3F6D3,0x354B9E99,0x4ECDAC24,
                   0x24C6B89E,0x72C0726F,0x58AA56F7};
const Ipp32u Or[]={0xa5a7939f,0x6ddebca3,0xd116bc4b,0x75d0fb98,
                   0x2a183025,0x26436686,0xd7c134aa};
IppsBigNumState* pP = nBN(sizeof(P)/4,P);
IppsBigNumState* pA = nBN(sizeof(A)/4,A);
IppsBigNumState* pB = nBN(sizeof(B)/4,B);
IppsBigNumState* pGX = nBN(sizeof(GX)/4,GX);
IppsBigNumState* pGY = nBN(sizeof(GY)/4,GY);
IppsBigNumState* pOr = nBN(sizeof(Or)/4,Or);
int cof=0x1;
ippsECCPSet(pP,pA,pB,pGX,pGY,pOr,cof,pECC);
IppsPRNGState* pRand=nPRNG();
int nTrials=20;
IppECResult pResult;
ippsECCPValidate(nTrials,&pResult,pECC,ippsPRNGen,pRand);
cout <<"EC Validation: "<<ippsECCGetResultString(pResult)<<endl;
const Ipp32u Priv[]={0x743734BE,0x04E5F2CE,0x4FC213CB,0xF212EB74,
                     0x7EB7A389,0x372E8025,0x2038D930};
const Ipp32u X[]= {0x72689DF2,0x6763F7ED,0x7277FA5F,0xDFDD9E5C,
                   0x8131DCC6,0x8416D4E8,0x37980C00};
const Ipp32u Y[]= {0x18CA56E1,0x3ABAA146,0x722A7540,0x1A5841E2,
                   0x208CBFC3,0xA1511C9D,0x349AB40D};
```

```
//Regular keys
IppsBigNumState* pPriv = nBN(sizeof(Priv)/4,Priv);
IppsBigNumState* pX = nBN(sizeof(X)/4,X);
IppsBigNumState* pY = nBN(sizeof(Y)/4,Y);
int pPointSize;
ippsECCPPointGetSize(bitS,&pPointSize);
IppsECCPPointState* pPub=
                 (IppsECCPPointState *)(new Ipp8u [pPointSize]);
ippsECCPPointInit(bitS,pPub);
ippsECCPSetPoint(pX,pY,pPub,pECC);
ippsECCPCheckPoint(pPub,&pResult,pECC);
cout << "CheckPoint: "<< ippsECCGetResultString(pResult)<<endl;
ippsECCPSetKeyPair(pPriv,pPub,ippTrue,pECC);
ippsECCPValidateKeyPair(pPriv,pPub,&pResult,pECC);
cout<<"RegKeyPairValidation: "<<ippsECCGetResultString(pResult)
                                                        <<endl;
//Ephemeral keys
IppsBigNumState* pephPrivate = nBN(bitS/32);
IppsECCPPointState* pephPublic=
                  (IppsECCPPointState *)(new Ipp8u [pPointSize]);
ippsECCPPointInit(bitS,pephPublic);
ippsECCPGenKeyPair(pephPrivate,pephPublic,pECC,ippsPRNGen,pRand);
ippsECCPSetKeyPair(pephPrivate,pephPublic,ippFalse,pECC);
ippsECCPPublicKey(pephPrivate,pephPublic,pECC);
ippsECCPValidateKeyPair(pephPrivate,pephPublic,&pResult,pECC);
cout <<"ephKeyPairValidation: "<<ippsECCGetResultString(pResult)
                                                         <<endl;
//Digest
Ipp32u dat[]={0x18CA56E1,0x3ABAA146,0x722A7540,0x1A5841E2,
              0x208CBFC3,0xA1511C9D,0x349AB40D};
IppsBigNumState* digest=nBN(bitS/32,dat);
IppsBigNumState* pSignX = nBN(bitS/32);
IppsBigNumState* pSignY = nBN(bitS/32);
//Signature
IppStatus w;
w=ippsECCPSignDSA(digest,pPriv,pSignX,pSignY,pECC);
cout << "Sign: "<<ippGetStatusString(w) << endl;
// Output: Sign: ippStsNoErr: No error, it's OK
```

```
//Verification
w=ippsECCPVerifyDSA(digest,pSignX,pSignY,&pResult,pECC);
cout <<"VerifyDSA: "<<ippsECCGetResultString(pResult)<<endl;
// Output: VerifyDSA:  Validation pass successfully
cout <<"Verify: "<< ippGetStatusString(w) << endl;
// Output: Verify: ippStsNoErr: No error, it's OK
delete [] (Ipp8u*)pSignX;
delete [] (Ipp8u*)pSignY;
delete [] (Ipp8u*)digest;
delete [] (Ipp8u*)pephPublic;
delete [] (Ipp8u*)pephPrivate;
delete [] (Ipp8u*)pPub;
delete [] (Ipp8u*)pPriv;
delete [] (Ipp8u*)pX;
delete [] (Ipp8u*)pY;
delete [] (Ipp8u*)pP;
delete [] (Ipp8u*)pA;
delete [] (Ipp8u*)pB;
delete [] (Ipp8u*)pGX;
delete [] (Ipp8u*)pGY;
delete [] (Ipp8u*)pOr;
delete [] (Ipp8u*)pRand;
delete [] (Ipp8u*)pECC;
return 0;
}
```

13.8 ECCP based Diffie-Hellman scheme

If two parties A, B share the same elliptic curve and have prepared their private and public keys: `PrivA, PubA, PrivB, PubB`, then they can define a shared secret key as the x coordinate of a point on the elliptic curve. The point is calculated as follows.

- B calculates: `PointShareB=PrivB*PubA`,
- A calculates: `PointShareA=PrivA*PubB`.

Since `PubA=PrivA*G` and `PubB=PrivB*G` both points have the same value `PrivA*PrivB*G` (`G` denotes the generator of the group of points on the elliptic curve).

```
IppStatus ippsECCPSharedSecretDH(const IppsBigNumState* Prv,
                                          //input:private key
         const IppsECCPPointState * Pub,   //input:public key
         const IppsBigNumState * Share,//output:shared secret
         IppsECCPState* Ctx)             //input:ECCP context
```

There is also a modified version `ECCPSharedSecretDHC`. In this case A, B compute the shared point with the use of the cofactor `Cof` of the elliptic curve.

- B calculates: `PointShareB=Cof*PrivB*PubA`,
- A calculates: `PointShareA=Cof*PrivA*PubB`.

```
IppStatus ippsECCPSharedSecretDHC(const IppsBigNumState* Prv,
                                          //input:private key
         const IppsECCPPointState * Pub,   //input:public key
         const IppsBigNumState * Share,//output:shared secret
         IppsECCPState* Ctx)             //input:ECCP context
```

```
// ECCP shared secret
// icpc -ipp=crypto 040ECCP08.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, nPRNG, nECCP, nECCPPoint
#include<ctime>

int main(){
using namespace std;
srand(time(0));
int  size;
int bitS=256;

// Standard EC 256r1 Setting
ippsECCPGetSize(bitS,&size);
IppsECCPState* ECCP=nECCP(bitS);
ippsECCPSetStd(IppECCPStd256r1,ECCP);
```

```
IppsPRNGState* Rand=nPRNG();
IppECResult Result;

// A Regular Keys
int PointSize;
ippsECCPPointGetSize(bitS,&PointSize);
IppsBigNumState* PrivA = nBN(bitS/32);
IppsECCPPointState* PubA=nECCPPoint(bitS,PointSize);
ippsECCPGenKeyPair(PrivA,PubA,ECCP,ippsPRNGen,Rand);
ippsECCPSetKeyPair(PrivA,PubA,ippTrue,ECCP);
ippsECCPValidateKeyPair(PrivA,PubA,&Result,ECCP);
cout<<"A regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    << endl;
// Output: A regKeyPairValidation: Validation pass successfully

// B Regular keys
IppsBigNumState* PrivB = nBN(bitS/32);
IppsECCPPointState* PubB=nECCPPoint(bitS,PointSize);
ippsECCPGenKeyPair(PrivB,PubB,ECCP,ippsPRNGen,Rand);
ippsECCPSetKeyPair(PrivB,PubB,ippTrue,ECCP);
ippsECCPValidateKeyPair(PrivB,PubB,&Result,ECCP);
cout<<"B regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    << endl;
// Output: B regKeyPairValidation: Validation pass successfully

// ECCPSharedSecretDH
IppsBigNumState* ShareA= nBN(bitS/32);
IppsBigNumState* ShareB= nBN(bitS/32);
ippsECCPSharedSecretDH(PrivA,PubB,ShareA,ECCP);
ippsECCPSharedSecretDH(PrivB,PubA,ShareB,ECCP);
//ippsECCPSharedSecretDHC(PrivA,PubB,ShareA,ECCP);
//ippsECCPSharedSecretDHC(PrivB,PubA,ShareB,ECCP);
Ipp32u Rsult;
ippsCmp_BN(ShareA, ShareB, &Rsult);
cout <<  Rsult<< " (result 0 ==> valid SharedSecretDH)";
// 0:equal 1:first greater 2:second greater
cout << "\n";
// Output: 0 (result 0 ==> valid SharedSecretDH)
```

```
delete [] (Ipp8u*)ShareA;
delete [] (Ipp8u*)ShareB;
delete [] (Ipp8u*)PubB;
delete [] (Ipp8u*)PrivB;
delete [] (Ipp8u*)PubA;
delete [] (Ipp8u*)PrivA;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)ECCP;
return 0;
}
```

Chapter 14

Elliptic curve cryptography over binary finite field

14.1 Binary finite field GF(2^m)

The symbol $GF(2^m)$ denotes the set of m-bit strings

$$a = (a_{m-1}, a_{m-2}, \ldots, a_0), \quad a_i \in \{0, 1\},$$

together with appropriately defined operations of addition and multiplication. To perform the addition of two m-bit strings it is sufficient to add the bits at the same positions modulo 2. In other words the addition of m-bit strings denotes component-wise XOR operation on the corresponding bits. To define the multiplication of m-bit strings it is easier to consider the bit-strings as polynomials with coefficients in $\{0, 1\}$

$$a(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0, \quad a_i \in \{0, 1\}.$$

As a part of definition of $GF(2^m)$ one introduces a polynomial of degree m

$$f(x) = f_m x^m + f_{m-1}x^{m-1} + \cdots + f_1x + f_0, \quad f_i \in \{0, 1\},$$

which is irreducible in the sense that it can not be expressed as an usual product of polynomials of lower degrees (with coefficients in $\{0, 1\}$). To compute the product of two elements a, b from $GF(2^m)$ it suffices to multiply the corresponding polynomials $a(x), b(x)$ in the usual way and next to compute the remainder from the division of $a(x) \cdot b(x)$ by the irreducible polynomial $f(x)$. Due to this definition the product of two elements of $GF(2^m)$ is still an element of $GF(2^m)$. Note that the coefficient arithmetic has to be performed modulo 2.

Example As the irreducible polynomial in $GF(2^{163})$ one can choose $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. Using the open-source package Sage it is easy to obtain the polynomial, the corresponding bit-string and its hexadecimal representation.

```
sage: pol=GF(2^163,'x').modulus();pol
x^163 + x^7 + x^6 + x^3 + 1
sage: f=(pol.change_ring(ZZ)(2))          # x->2
sage: bin(f)
'0b1000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000
00000000000000011001001'
sage: hex(f)
'800000000000000000000000000000000000c9'
```

In examples we shall use the hexadecimal representation of the modulus (irreducible polynomial) and the elements of $GF(2^m)$.
The conversion from hexadecimal representation to polynomial can be performed as follows.

```
sage: Z.<x>=ZZ[]
sage: Z(0x800000000000000000000000000000000000c9.digits(2))
x^163 + x^7 + x^6 + x^3 + 1
```

14.2 Creating a new ECCB context

Before we start to use the elliptic curves over binary fields we need to prepare an appropriate context. A new ECCB context can be created with the help of `ECCBGetSize` and `ECCBInit`.

```
IppStatus ippsECCBGetSize(  int bitS, //input:curve bit-size
                            int* size)  //output:ECCB context
                                        //size in bytes
```

The size obtained from `ECCBGetSize` can be used to allocate a sufficient amount of memory for the ECCB context. The memory can be initialized with the use of `ECCBInit`.

```
IppStatus ippsECCBInit(int bitS,       //input:curve bit-size
                   IppsECCBState* ECCB)//output:ECCB context
```

In examples we shall use the function `nECCB` which performs the just mentioned steps.

```
IppsECCBState* nECCB(int bitS)
{ int size;
  ippsECCBGetSize(bitS,&size);
  IppsECCBState* ECCB=(IppsECCBState *)(new Ipp8u [size]);
  ippsECCBInit(bitS,ECCB);
  return ECCB;
}
```

14.3 Standard $\mathbf{GF(2^m)}$ elliptic curve setup

The shortest way to set the parameters for the ECCB context is to use the function `ECCBSetStd`.

```
IppStatus ippsECCBSetStd(IppECCType flag,  //input:curve flag
                   IppsECCBState* ECCB)//output:ECCB context
```

The `flag` can take one of the following values:

```
IppECCBStd113r1,    IppECCBStd113r2,    IppECCBStd131r1,
IppECCBStd131r2,    IppECCBStd163k1,    IppECCBStd163r1,
IppECCBStd163r2,    IppECCBStd193r1,    IppECCBStd193r2,
IppECCBStd233k1,    IppECCBStd233r1,    IppECCBStd239k1,
IppECCBStd283k1,    IppECCBStd283r1,    IppECCBStd409k1.
```

The parameters of the elliptic curve over $GF(2^m)$ can be verified with the help of `ECCBValidate`.

```
IppStatus ippsECCBValidate( int Trials, //input:numb.of tests
                IppECResult* Result, //output:validation res.
                IppsECCBState* Ctx,      //input:ECCB context
                IppBitSupplier PRNGen,//input:pseudorand.gen.
                void* Rand)    //input:pseudorand gen.context
```

All the introduced functions are used in the following simple example.

```
// Definition and validation of the standard ECCB
//icpc -xHOST -ipp=crypto   040ECCB00.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
IppsECCBState* ECCB=nECCB(163);
ippsECCBSetStd(IppECCBStd163k1,ECCB);
IppsPRNGState* Rand=nPRNG();
int nTrials=20;
IppECResult Result;
ippsECCBValidate(nTrials,&Result,ECCB,ippsPRNGen,Rand);
cout<<"EC Validation: "<<ippsECCGetResultString(Result)<<endl;
delete [] (Ipp8u*)ECCB;
delete [] (Ipp8u*)Rand;
return 0;
}
//Output:
//EC Validation: Validation pass successfully
```

14.4 Parameters of the elliptic curve over binary finite field

An elliptic curve `E` over $GF(2^m)$ is defined by the equation

$$\mathtt{y}^2 + \mathtt{x} \cdot \mathtt{y} = \mathtt{x}^3 + \mathtt{A} \cdot \mathtt{x}^2 + \mathtt{B} \qquad \texttt{in GF}(2^{\mathtt{m}}),$$

so to define such a curve it is necessary to specify the `BigNum` parameters `P`, `A`, `B`, where `P` is the irreducible polynomial (modulus) defining the finite field and `A`, `B` are the elliptic curve coefficients. Additionally we have to set the `BigNum` x, y coordinates `GX, GY` of the base point `G`, i.e. the point generating some subgroup of points on the curve. It is also necessary to set the `BigNum` - the `Order` of the group generated by `G` and the cofactor `Cof`, i.e. the quotient #E/`Order`, where #E denotes the number of all points on `E`. All the parameters can be set with the help of `ECCBSet`.

```
IppStatus ippsECCBSet(const IppsBigNumState* P,//input:modulus
          const IppsBigNumState* A,           //input:coeff.A
          const IppsBigNumState* B,           //input:coeff.B
          const IppsBigNumState* GX,//input:X-coordinate of G
          const IppsBigNumState* GY,//input:Y-coordinate of G
          const IppsBigNumState* Ord,     //input:Group order
          int Cof,                          //input:cofactor
          IppsECCBState* Ctx)           //output:ECCB context
```

If the elliptic curve is already defined then all its parameters can be retrieved from the context with the use of `ECCBGet`.

```
IppStatus ippsECCBGet(IppsBigNumState* P,    //output:modulus
               IppsBigNumState* A,           //output:coeff.A
               IppsBigNumState* B,           //output:coeff.B
               IppsBigNumState* GX,//output:X-coordinate of G
               IppsBigNumState* GY,//output:Y-coordinate of G
               IppsBigNumState* Ord,     //output:Group order
               int Cof,                     //output:cofactor
               IppsECCBState* Ctx)       //input:ECCB context
```

```
 IppStatus ippsECCBGetOrderBitSize(int bitS, //output:curve
                                             //bit-size
                   IppsECCBState* ECC)//input:ECCB context
```

In the first example we use a standard elliptic curve.

```
// Retrieving parameters of the standard EC over binary field
// icpc -xHOST -ipp=crypto   040ECCB01.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=163;
IppsECCBState* ECCB=nECCB(bitS);
ippsECCBSetStd(IppECCBStd163r1,ECCB);
IppsBigNumState* P = nBN(bitS/32+1);// Modulus P(irr.polyn.)
IppsBigNumState* A = nBN(bitS/32+1);// Coeff A
```

```
IppsBigNumState* B = nBN(bitS/32+1);// Coeff B
IppsBigNumState* GX = nBN(bitS/32+1);// Base Point Coord. X
IppsBigNumState* GY = nBN(bitS/32+1);// Base Point Coord. Y
int bitSize;
ippsECCBGetOrderBitSize(&bitSize,ECCB);
IppsBigNumState* Order = nBN(bitSize/32+1);//BasePoint Ord.
int  cofactor;                          // cofactor=#E/Order
// Getting EC parameters
ippsECCBGet(P,A,B,GX,GY,Order,&cofactor,ECCB);
tBN("P:\n",P);
// P:00000008000000000000000000000000000000000000000c9
tBN("A:\n",A);
// A:00000007b6882caaefa84f9554ff8428bd88e246d2782ae2
tBN("B:\n",B);
// B:0000000713612dcddcb40aab946bda29ca91f73af958afd9
tBN("Order:\n",Order);
// Order:00000003ffffffffffffffffff48aab689c29ca710279b
cout<<"Cof: "<<cofactor<<endl;
// Cof: 2
delete [] (Ipp8u*)Order;
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)A;
delete [] (Ipp8u*)B;
delete [] (Ipp8u*)GX;
delete [] (Ipp8u*)GY;
delete [] (Ipp8u*)ECCB;
return 0;
}
```

In the second example we explicitly set all the parameters (known from the previous example).

```
//Setting EC parameters and EC validation
//icpc -xHOST  -ipp=crypto  040ECCB02.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
```

```
int main(){
using namespace std;
int bitS=163;
IppsECCBState* ECC=nECCB(bitS);
const Ipp32u P[]= {0x000000c9,0x00000000,0x00000000,
                   0x00000000,0x00000000,0x00000008};
const Ipp32u A[]= {0xd2782ae2,0xbd88e246,0x54ff8428,
                   0xefa84f95,0xb6882caa,0x00000007};
const Ipp32u B[]= {0xf958afd9,0xca91f73a,0x946bda29,
                   0xdcb40aab,0x13612dcd,0x00000007};
const Ipp32u GX[]={0x7876a654,0x567f787a,0x89566789,
                   0xab438977,0x69979697,0x00000003};
const Ipp32u GY[]={0xf41ff883,0xe3c80988,0x9d51fefc,
                   0xefafb298,0x435edb42,0x00000000};
const Ipp32u Or[]={0xa710279b,0xb689c29c,0xffff48aa,
                   0xffffffff,0xffffffff,0x00000003};
IppsBigNumState* pP = nBN(sizeof(P)/4,P);
IppsBigNumState* pA = nBN(sizeof(A)/4,A);
IppsBigNumState* pB = nBN(sizeof(B)/4,B);
IppsBigNumState* pGX = nBN(sizeof(GX)/4,GX);
IppsBigNumState* pGY = nBN(sizeof(GY)/4,GY);
IppsBigNumState* pOr = nBN(sizeof(Or)/4,Or);
int cof=0x2;
ippsECCBSet(pP,pA,pB,pGX,pGY,pOr,cof,ECC);
IppsPRNGState* Rand=nPRNG();
int Trials=20;
IppECResult Result;
ippsECCBValidate(Trials,&Result,ECC,ippsPRNGen,Rand);
cout<<"EC Validation: "<<ippsECCGetResultString(Result)<<endl;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)pP;
delete [] (Ipp8u*)pA;
delete [] (Ipp8u*)pB;
delete [] (Ipp8u*)pGX;
delete [] (Ipp8u*)pGY;
delete [] (Ipp8u*)pOr;
delete [] (Ipp8u*)ECC;
return 0;
```

```
}
// Output:
// EC Validation: Validation pass successfully
```

14.5 Points on the elliptic curve over binary field

The points on an elliptic curve over binary field have a context which must be prepared and initialized. The evaluation of context size can be done with the help of `ECCBPointGetSize`.

```
IppStatus ippsECCBPointGetSize(int bitS,//input:curve bit-size
                           int* Psize)  //output:EC point size
```

The appropriate amount of memory must be allocated and initialized.

```
IppStatus ippsECCBPointInit(int bitS,   //input:curve bit-size
                    IppsECCBPointState* P)//output:initialized
                                          //EC point context
```

In examples we shall define new points contexts using the function `nECCBPoint`.

```
IppsECCBPointState* nECCBPoint(int bitS,int Psize)
{
IppsECCBPointState* Point=(IppsECCBPointState *)
                                          (new Ipp8u [Psize]);
ippsECCBPointInit(bitS,Point);
return Point;
}
```

The coordinates of the point can be set with the use of `ECCBSetPoint`.

```
IppStatus ippsECCBSetPoint(const IppsBigNumState* X,//input:
                                                  X-coordinate
              const IppsBigNumState* Y, //input:Y-coordinate
              IppsECCBPointState* P,       //output:EC point
              IppsECCBState* ECCB)      //input:ECCB context
```

There is a special function defining the point at infinity.

```
IppStatus ippsECCBSetPointAtInfinity(IppsECCBPointState* O,
                                     //output:point at infinity
                    IppsECCBState* ECCB)//input:ECCB context
```

The point can be validated using the function `ECCBCheckPoint`.

```
 IppStatus ippsECCBCheckPoint(const IppsECCBPointState* P,
                                            //input:EC point
              IppECResult* Result,//output:validation result
              IppsECCBState* ECCB)      //input:ECCB context
```

The x, y coordinates can be retrieved with the use of `ECCBGetPoint`.

```
 IppStatus ippsECCBGetPoint(IppsBigNumState* X,    //output:
                                                 X-coordinate
                   IppsBigNumState* Y,//output:Y-coordinate
                   IppsECCBPointState* P,  //input:EC point
                   IppsECCBState* ECCB)//input:ECCB context
```

```
//Definition and validation of the point on EC over binary field
//icpc -xHOST -ipp=crypto   040ECCB03.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=163;
IppsECCBState* ECCB=nECCB(bitS);
ippsECCBSetStd(IppECCBStd163r1,ECCB);
int Psize;
ippsECCBPointGetSize(bitS,&Psize);
IppsECCBPointState* P=nECCBPoint(bitS,Psize);
const Ipp32u GX[]={0x7876a654,0x567f787a,0x89566789,
                   0xab438977,0x69979697,0x00000003};
const Ipp32u GY[]={0xf41ff883,0xe3c80988,0x9d51fefc,
                   0xefafb298,0x435edb42,0x00000000};
IppsBigNumState* X = nBN(bitS/32+1,GX);
IppsBigNumState* Y = nBN(bitS/32+1,GY);
ippsECCBSetPoint(X,Y,P,ECCB);
```

```
IppsBigNumState* X1 = nBN(bitS/32+1,GX);
IppsBigNumState* Y1 = nBN(bitS/32+1,GY);
ippsECCBGetPoint(X1,Y1,P,ECCB);
tBN("X: ",X1);
IppECResult Result;
ippsECCBCheckPoint(P,&Result,ECCB);
cout << "CheckPoint: "<< ippsECCGetResultString(Result) << endl;
delete [] (Ipp8u*)X1;
delete [] (Ipp8u*)Y1;
delete [] (Ipp8u*)X;
delete [] (Ipp8u*)Y;
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)ECCB;
return 0;
}
//Output:
//X:0000000369979697ab43897789566789567f787a7876a654
//CheckPoint: Validation pass successfully
```

14.6 Arithmetic of elliptic curves over binary finite fields

As in the case of curves over prime fields we have at our disposal four functions.

1. The comparison of two points on an elliptic curve over finite binary field can be performed with the help of `ECCBComparePoint`.

```
IppStatus ippsECCBComparePoint(const IppsECCBPointState* P,
                                          //input:EC point
             const IppsECCBPointState* Q,  //input:EC point
             IppECResult* Result,//output:comparison result
             IppsECCBState* ECCB)      //input:ECCB context
```

2. The negation (-P) of a point on an elliptic curve can be obtained with the help of `ECCBNegativePoint`.

```
IppStatus ippsECCBNegativePoint(const IppsECCBPointState* P,
                                                  //input:EC point
                    IppsECCBPointState* Q,//output:EC point -P
                    IppsECCBState* ECCB)  //input:ECCB context
```

3. The addition of two points on an elliptic curve can be performed by the function `ECCBAddPoint`.

```
 IppStatus ippsECCBAddPoint(const IppsECCBPointState* P,
                                                  //input:EC point
                const IppsECCBPointState* Q, //input:EC point
                IppsECCBPointState* R,//output:EC point R=P+Q
                IppsECCBState* ECCB)     //input:ECCB context
```

4. The multiplication of a point on an elliptic curve by a `BigNum` can be accomplished by `ECCBMulPointScalar`.

```
IppStatus ippsECCBMulPointScalar(const IppsECCBPointState* P,
                                                  //input:EC point
            const IppsBigNumState* M,//input:BigNum multiplier
            IppsECCBPointState* R,     //output:EC point R=M*P
            IppsECCBState* ECCB)          //input:ECCB context
```

```
// EC arithmetic over binary finite field
//icpc -xHOST -ipp=crypto   040ECCB04.cpp toolb.cpp
#include "ippcp.h"
#include <iostream>
#include "toolb.h"
int main(){
using namespace std;
int bitS=163;
IppsECCBState* ECCB=nECCB(bitS);
ippsECCBSetStd(IppECCBStd163r1,ECCB);
int Psize;
ippsECCBPointGetSize(bitS,&Psize);
IppsECCBPointState* P=nECCBPoint(bitS,Psize);
const Ipp32u GX[]={0x7876a654,0x567f787a,0x89566789,
                   0xab438977,0x69979697,0x00000003};
const Ipp32u GY[]={0xf41ff883,0xe3c80988,0x9d51fefc,
```

```
                    0xefafb298,0x435edb42,0x00000000};
IppsBigNumState* X = nBN(bitS/32+1,GX);
IppsBigNumState* Y = nBN(bitS/32+1,GY);
ippsECCBSetPoint(X,Y,P,ECCB);
IppsECCBPointState* R=nECCBPoint(bitS,Psize);
ippsECCBNegativePoint(P,R,ECCB);
IppsECCBPointState* Q=nECCBPoint(bitS,Psize);
ippsECCBAddPoint(P,R,Q,ECCB);
IppsECCBPointState* O=nECCBPoint(bitS,Psize);
ippsECCBSetPointAtInfinity(O,ECCB);
IppECResult Result;
ippsECCBComparePoint(Q,O,&Result,ECCB);
cout << "P+(-P)==O: "<< ippsECCGetResultString(Result) << endl;
int bitSize;
ippsECCBGetOrderBitSize(&bitSize,ECCB);
IppsBigNumState* Ord = nBN(bitSize/32+1);
IppsBigNumState* X1 = nBN(bitS/32+1);
IppsBigNumState* Y1 = nBN(bitS/32+1);
int cof;
ippsECCBGet(X1,X1,Y1,X1,Y1,Ord,&cof,ECCB);
ippsECCBMulPointScalar(P,Ord,R,ECCB);
ippsECCBComparePoint(O,R,&Result,ECCB);
cout << "P^Ord==O: "<< ippsECCGetResultString(Result) << endl;
delete [] (Ipp8u*)X1;
delete [] (Ipp8u*)Y1;
delete [] (Ipp8u*)Ord;
delete [] (Ipp8u*)O;
delete [] (Ipp8u*)P;
delete [] (Ipp8u*)Q;
delete [] (Ipp8u*)R;
delete [] (Ipp8u*)X;
delete [] (Ipp8u*)Y;
delete [] (Ipp8u*)ECCB;
return 0;
}
//Output:
//P+(-P)==O: Points are equal
//P^Ord==O: Points are equal
```

14.7 ECCB cryptosystem keys

To use the ECCB cryptographic primitives we have to establish private and public keys.

- The private key `Prv` is a `BigNum` in the interval `[1,Ord-1]`, where `Ord` is the order of the base point `G` (generating the group of points under consideration). The bit-size of `Prv` must be less than the parameter retrieved by the function `ECCBGetOrderSize`.
- The public key `Pub` is the `ECCBPoint` of the form `Prv*G`.

The keys can be generated with the use of the function `ECCBGenKeyPair`.

```
IppStatus ippsECCBGenKeyPair( IppsBigNumState * Prv, //output:
                                                    //private key
                  IppsECCBPointState* Pub,  //output:public key
                  IppsECCBState* Ctx,      //input:ECCB context
                  IppBitSupplier PRNGen,//input:pseudorand.gen.
                  void* Rand)    //input:pseudorand gen.context
```

The public key can be computed from the given private key. The computation can be performed by the function `ECCBPublicKey`.

```
 IppStatus ippsECCBPublicKey( const IppsBigNumState * Prv,
                                          //input:private key
                   IppsECCBPointState* Pub,//output:public key
                   IppsECCBState* Ctx)    //input:ECCB context
```

The obtained keys can be validated with the help of `ECCBValidateKeyPair`.

```
IppStatus ippsECCBValidateKeyPair(const IppsBigNumState * Prv,
                                           //input:private key
              const IppsECCBPointState* Pub,//input:public key
              IppECResult* Reslt,   //output:validation result
              IppsECCBState* Ctx)         //input:ECCB context
```

The keys can be set with the use of the function `ECCBSetKeyPair`.

```
IppStatus ippsECCBSetKeyPair(const IppsBigNumState * Prv,
                                          //input:private key
            const IppsECCBPointState* Pub,//input:public key
            IppBool KeyType,               //input:key-type
            IppsECCBState* Ctx)         //input:ECCB context
//KeyType values:  ippTrue (regular) or ippFalse (ephemeral)
```

```
// ECCB Keys
// icpc -ipp=crypto 040ECCB05.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, nPRNG, nECCB, nECCBPoint
#include <ctime>

int main(){
using namespace std;
srand(time(0));
int  size;
int bitS=163;
// Standard ECCB 163r1 Setting
ippsECCBGetSize(bitS,&size);
IppsECCBState* ECCB=nECCB(bitS);
ippsECCBSetStd(IppECCBStd163r1,ECCB);
// Regular Keys
int PointSize;
ippsECCBPointGetSize(bitS,&PointSize);
IppsBigNumState* Priv = nBN(bitS/32+1);
IppsECCBPointState* Pub=nECCBPoint(bitS,PointSize);
IppsPRNGState* Rand=nPRNG();
ippsECCBGenKeyPair(Priv,Pub,ECCB,ippsPRNGen,Rand);
ippsECCBSetKeyPair(Priv,Pub,ippTrue,ECCB);
IppECResult Result;
ippsECCBValidateKeyPair(Priv,Pub,&Result,ECCB);
cout<<"regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    << endl;
// Output: regKeyPairValidation: Validation pass successfully
// Ephemeral Keys
IppsBigNumState* ephPriv = nBN(bitS/32+1);
```

```
IppsECCBPointState* ephPub=nECCBPoint(bitS,PointSize);
ippsECCBGenKeyPair(ephPriv,ephPub,ECCB,ippsPRNGen,Rand);
ippsECCBSetKeyPair(ephPriv,ephPub,ippFalse,ECCB);
ippsECCBPublicKey(ephPriv,ephPub,ECCB);
ippsECCBValidateKeyPair(ephPriv,ephPub,&Result,ECCB);
cout<<"ephKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    << endl;
// Output: ephKeyPairValidation: Validation pass successfully
delete [] (Ipp8u*)ephPriv;
delete [] (Ipp8u*)ephPub;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)Priv;
delete [] (Ipp8u*)Pub;
delete [] (Ipp8u*)ECCB;
return 0;
}
```

14.8 ECCB based digital signature

A digital signature of a message consists of two `BigNums`, say `SignX` and `SignY`. To compute the signature, we first have to apply a hash function to the message. As a result we obtain the message digest. The message digest and user's private key allow for creating user's signature. The appropriate procedure is performed by the function `ECCBSignDSA`.

```
IppStatus ippsECCBSignDSA(const IppsBigNumState* Digest,
                                        //input:message digest
            const IppsBigNumState * Prv, //input:private key
            IppsBigNumState * SignX,//output:X comp.of sign.
            IppsBigNumState * SignY,//output:Y comp.of sign.
            IppsECCBState* Ctx)         //input:ECCB context
```

The message digest and user's regular public key are used in the verification procedure which is performed by the function `ECCBVerifyDSA`.

```
IppStatus ippsECCBVerifyDSA(const IppsBigNumState* Digest,
                                          //input:message digest
         const IppsBigNumState * SignX,//input:X comp.of sign.
         const IppsBigNumState * SignY,//input:Y comp.of sign.
         IppsECCBState* Ctx)              //input:ECCB context
```

There are also Nyberg-Rueppel versions of the signature scheme.

```
 IppStatus ippsECCBSignNR(const IppsBigNumState* Digest,
                                         //input:message digest
             const IppsBigNumState * Prv, //input:private key
             IppsBigNumState * SignX,//output:X comp.of sign.
             IppsBigNumState * SignY,//output:Y comp.of sign.
             IppsECCBState* Ctx)         //input:ECCB context
```

```
IppStatus ippsECCBVerifyNR(const IppsBigNumState* Digest,
                                          //input:message digest
         const IppsBigNumState * SignX,//input:X comp.of sign.
         const IppsBigNumState * SignY,//input:Y comp.of sign.
         IppsECCBState* Ctx)              //input:ECCB context
```

```
// ECCB DSA/NR signature
// icpc -ipp=crypto 040ECCB06.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, nPRNG, nECCB, nECCBPoint
#include <ctime>

int main(){
using namespace std;
srand(time(0));
int  size;
int bitS=163;
// Standard ECCB 163r1 setting
ippsECCBGetSize(bitS,&size);
IppsECCBState* ECCB=nECCB(bitS);
ippsECCBSetStd(IppECCBStd163r1,ECCB);
```

```
// Regular Keys
int PointSize;
ippsECCBPointGetSize(bitS,&PointSize);
IppsBigNumState* Priv = nBN(bitS/32+1);
IppsECCBPointState* Pub=nECCBPoint(bitS,PointSize);
IppsPRNGState* Rand=nPRNG();
ippsECCBGenKeyPair(Priv,Pub,ECCB,ippsPRNGen,Rand);
ippsECCBSetKeyPair(Priv,Pub,ippTrue,ECCB);
IppECResult Result;
ippsECCBValidateKeyPair(Priv,Pub,&Result,ECCB);
cout<<"regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    <<endl;
// Output: regKeyPairValidation: Validation pass successfully
// Ephemeral Keys
IppsBigNumState* ephPriv = nBN(bitS/32+1);
IppsECCBPointState* ephPub=nECCBPoint(bitS,PointSize);
ippsECCBGenKeyPair(ephPriv,ephPub,ECCB,ippsPRNGen,Rand);
ippsECCBSetKeyPair(ephPriv,ephPub,ippFalse,ECCB);
ippsECCBPublicKey(ephPriv,ephPub,ECCB);
ippsECCBValidateKeyPair(ephPriv,ephPub,&Result,ECCB);
cout<<"ephKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    <<endl;
// Output: ephKeyPairValidation: Validation pass successfully
// Digest
Ipp32u ddat[] = {
      0xBEAD208B, 0x5E668076, 0x2ABF62E3, 0xDB7C000};
IppsBigNumState* digest=nBN(bitS/32+1,ddat);
// Signature
IppsBigNumState* SignX = nBN(bitS/32+1);
IppsBigNumState* SignY = nBN(bitS/32+1);
IppStatus w;
w=ippsECCBSignDSA(digest,Priv,SignX,SignY,ECCB);
//w=ippsECCBSignNR(digest,Priv,SignX,SignY,ECCB);
cout << "Sign: "<<ippGetStatusString(w) << endl;
// Output: Sign: ippStsNoErr: No error, it's OK
//Verification
w=ippsECCBVerifyDSA(digest,SignX,SignY,&Result,ECCB);
//w=ippsECCBVerifyNR(digest,SignX,SignY,&Result,ECCB);
```

```
cout << "Verify DSA: "<< ippsECCGetResultString(Result) << endl;
// Output: Verify DSA:  Validation pass successfully
cout <<"Verify: "<< ippGetStatusString(w) << endl;
// Output: Verify: ippStsNoErr: No error, it's OK
delete [] (Ipp8u*)SignX;
delete [] (Ipp8u*)SignY;
delete [] (Ipp8u*)digest;
delete [] (Ipp8u*)ephPub;
delete [] (Ipp8u*)ephPriv;
delete [] (Ipp8u*)Pub;
delete [] (Ipp8u*)Priv;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)ECCB;
return 0;
}
```

14.9 ECCB based Diffie-Hellman scheme

If two parties A, B share the same elliptic curve and have prepared their private and public keys: `PrivA, PubA, PrivB, PubB`, then they can define a shared secret key as the x coordinate of a point on the elliptic curve. The point is calculated as follows.

- B calculates: `PointShareB=PrivB*PubA`,
- A calculates: `PointShareA=PrivA*PubB`.

Since `PubA=PrivA*G` and `PubB=PrivB*G` both points have the same value `PrivA*PrivB*G` (`G` denotes the generator of the group of points on the elliptic curve).

```
IppStatus ippsECCBSharedSecretDH(const IppsBigNumState* Prv,
                                                //input:private key
          const IppsECCBPointState * Pub,   //input:public key
          const IppsBigNumState * Share,//output:shared secret
          IppsECCBState* Ctx)             //input:ECCB context
```

There is also a modified version `ECCBSharedSecretDHC`. In this case A, B compute the shared point with the use of the cofactor `Cof` of the elliptic curve.

- B calculates: `PointShareB=Cof*PrivB*PubA`,
- A calculates: `PointShareA=Cof*PrivA*PubB`.

```
IppStatus ippsECCBSharedSecretDHC(const IppsBigNumState* Prv,
                                          //input:private key
         const IppsECCBPointState * Pub,   //input:public key
         const IppsBigNumState * Share,//output:shared secret
         IppsECCBState* Ctx)             //input:ECCB context
```

```
// ECCB shared secret
// icpc -ipp=crypto 040ECCB08.cpp toolb.cpp
#include "ippcore.h"
#include "ippcp.h"
#include <iostream>
#include "toolb.h" // nBN, nPRNG, nECCB, nECCBPoint
#include<ctime>

int main(){
using namespace std;
srand(time(0));
int  size;
int bitS=163;

// Standard EC 163r1 setting
ippsECCBGetSize(bitS,&size);
IppsECCBState* ECCB=nECCB(bitS);
ippsECCBSetStd(IppECCBStd163r1,ECCB);

IppsPRNGState* Rand=nPRNG();
IppECResult Result;

// A Regular Keys
int PointSize;
ippsECCBPointGetSize(bitS,&PointSize);
IppsBigNumState* PrivA = nBN(bitS/32+1);
IppsECCBPointState* PubA=nECCBPoint(bitS,PointSize);
ippsECCBGenKeyPair(PrivA,PubA,ECCB,ippsPRNGen,Rand);
ippsECCBSetKeyPair(PrivA,PubA,ippTrue,ECCB);
```

```
ippsECCBValidateKeyPair(PrivA,PubA,&Result,ECCB);
cout<<"A regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    << endl;
// Output: A regKeyPairValidation: Validation pass successfully

// B Regular keys
IppsBigNumState* PrivB = nBN(bitS/32+1);
IppsECCBPointState* PubB=nECCBPoint(bitS,PointSize);
ippsECCBGenKeyPair(PrivB,PubB,ECCB,ippsPRNGen,Rand);
ippsECCBSetKeyPair(PrivB,PubB,ippTrue,ECCB);
ippsECCBValidateKeyPair(PrivB,PubB,&Result,ECCB);
cout<<"B regKeyPairValidation: "<<ippsECCGetResultString(Result)
                                                    << endl;
// Output: B regKeyPairValidation: Validation pass successfully

// ECCBSharedSecretDH
IppsBigNumState* ShareA= nBN(bitS/32+1);
IppsBigNumState* ShareB= nBN(bitS/32+1);
ippsECCBSharedSecretDH(PrivA,PubB,ShareA,ECCB);
ippsECCBSharedSecretDH(PrivB,PubA,ShareB,ECCB);
//ippsECCBSharedSecretDHC(PrivA,PubB,ShareA,ECCB);
//ippsECCBSharedSecretDHC(PrivB,PubA,ShareB,ECCB);
Ipp32u Rsult;
ippsCmp_BN(ShareA, ShareB, &Rsult);
cout <<  Rsult<< " (result 0 ==> valid SharedSecretDH)";
// 0:equal 1:first greater 2:second greater
cout << "\n";
// Output: 0 (result 0 ==> valid SharedSecretDH)
delete [] (Ipp8u*)ShareA;
delete [] (Ipp8u*)ShareB;
delete [] (Ipp8u*)PubA;
delete [] (Ipp8u*)PubB;
delete [] (Ipp8u*)PrivA;
delete [] (Ipp8u*)PrivB;
delete [] (Ipp8u*)Rand;
delete [] (Ipp8u*)ECCB;
return 0;
}
```

Appendix A

Auxiliary functions

In this appendix we define the functions that are used in examples but are not a part of IPP.

A.1 toolb.h

```
#if !defined _TOOLB_H_
#define _TOOLB_H_
#include "ippcp.h"
#include <stdlib.h>
Ipp32u* randIpp32u(Ipp32u* X, int size);
IppsBigNumState* nBN(int dS, const Ipp32u* A=0);
IppsBigNumState* nBNneg(int dS, const Ipp32u* A=0);
IppsPRNGState* nPRNG(int seedBitsize=160);
IppsPrimeState* nPrimG(int ,IppsPRNGState*);
IppsRSAState* nRSA(int NbitS, int PbitS, IppRSAKeyType type);
IppsDLPState* nDLP(int PbitS, int QbitS);
IppsECCPState* nECCP(int bitS);
IppsECCBState* nECCB(int bitS);
IppsECCPPointState* nECCPPoint(int bitS,int Psize);
IppsECCBPointState* nECCBPoint(int bitS,int Psize);
void tBN(const char* Msg, const IppsBigNumState* BN);
#endif // _TOOLB_H_
```

A.2 toolb.cpp

```
#include "toolb.h"
#include<iostream>

// new BN
IppsBigNumState* nBN(int dS, const Ipp32u* A)
                    // new (positive) BigNum
{                   // dS - data Size in Ipp32u words,
   int cS;          // A - data Array, cS - BigNum contex Size
   ippsBigNumGetSize(dS, &cS);// get the BigNum context Size
   IppsBigNumState* BN = (IppsBigNumState*)( new Ipp8u [cS] );
   ippsBigNumInit(dS, BN);    // BigNum context initialization
   if(A)
      ippsSet_BN(IppsBigNumPOS, dS, A, BN); // set the  sign
   return BN;                                // and the value
}

// new negative BN
IppsBigNumState* nBNneg(int dS, const Ipp32u* A)
{
   int size;
   ippsBigNumGetSize(dS, &size);
   IppsBigNumState* BN = (IppsBigNumState*)( new Ipp8u [size] );
   ippsBigNumInit(dS, BN);
   if(A)
      ippsSet_BN(IppsBigNumNEG, dS, A, BN);
   return BN;
}

// array of Ipp32u  random numbers, using rand() from stdlib
Ipp32u* randIpp32u(Ipp32u* X, int size)
{
   for(int n=0; n<size; n++)
      X[n] = (rand()<<16) + rand();
   return X;
}
```

```
// new PRNG, with default values of modulus, initial hash,
//                                          entropy augment
IppsPRNGState* nPRNG(int sBits)
{
   int sSize =(sBits+31)>>5;
   Ipp32u* seed = new Ipp32u [sSize];
//   Ipp32u* augm = new Ipp32u [sSize];
//   Ipp32u* h0   = new Ipp32u [sSize];
//   Ipp32u* mod   = new Ipp32u [sSize];
   int bnSize;
   IppsBigNumState* Tmp;
   ippsPRNGGetSize(&bnSize);
   IppsPRNGState* Ctx = (IppsPRNGState*)( new Ipp8u [bnSize] );
   ippsPRNGInit(sBits, Ctx);
   ippsPRNGSetSeed(Tmp=nBN(sSize,randIpp32u(seed,sSize)), Ctx);
//ippsPRNGSetAugment(Tmp=nBN(sSize,randIpp32u(augm,sSize)),Ctx);
//delete [] (Ipp8u*)Tmp;
//ippsPRNGSetH0(Tmp=nBN(sSize,randIpp32u(h0,sSize)),Ctx);
//delete [] (Ipp8u*)Tmp;
//ippsPRNGSetModulus(Tmp=nBN(sSize,randIpp32u(mod,sSize)),Ctx);
   delete [] (Ipp8u*)Tmp;
   delete [] seed;
//   delete [] augm;
//   delete [] h0;
//   delete [] mod;
   return Ctx;
}

// new prime gen
IppsPrimeState* nPrimG(int Bits,IppsPRNGState* Rand)
{
int size;
ippsPrimeGetSize(Bits, &size);
IppsPrimeState* PrimCtx = (IppsPrimeState*)( new Ipp8u [size] );
ippsPrimeInit(Bits, PrimCtx);
return PrimCtx;
}
```

```
// new RSA
IppsRSAState* nRSA(int NbitS, int PbitS, IppRSAKeyType Ktype)
{
   int size;
   ippsRSAGetSize(NbitS,PbitS, Ktype, &size);
   IppsRSAState* Ctx = (IppsRSAState*)( new Ipp8u [size] );
   ippsRSAInit(NbitS,PbitS, Ktype, Ctx);
   return Ctx;
}

// new DLP
IppsDLPState* nDLP(int PbitS, int QbitS)
{
   int size;
   ippsDLPGetSize(PbitS, QbitS, &size);
   IppsDLPState *Ctx= (IppsDLPState *)new Ipp8u[ size ];
   ippsDLPInit(PbitS, QbitS, Ctx);
   return Ctx;
}

// new ECCP
IppsECCPState* nECCP(int bitS)
{
  int size;
  ippsECCPGetSize(bitS,&size);
  IppsECCPState* ECCP=(IppsECCPState *)(new Ipp8u [size]);
  ippsECCPInit(bitS,ECCP);
  return ECCP;
}

// new ECCB
IppsECCBState* nECCB(int bitS)
{
  int size;
  ippsECCBGetSize(bitS,&size);
  IppsECCBState* ECCB=(IppsECCBState *)(new Ipp8u [size]);
  ippsECCBInit(bitS,ECCB);
  return ECCB;
```

```
}

// new ECCP point
IppsECCPPointState* nECCPPoint(int bitS,int Psize)
{ IppsECCPPointState* Point=(IppsECCPPointState *)
                                          (new Ipp8u [Psize]);
  ippsECCPPointInit(bitS,Point);
  return Point;
}

// new ECCB point
IppsECCBPointState* nECCBPoint(int bitS,int Psize)
{ IppsECCBPointState* Point=(IppsECCBPointState *)
                                           (new Ipp8u [Psize]);
  ippsECCBPointInit(bitS,Point);
  return Point;
}

// type BN
void tBN(const char* Msg,const IppsBigNumState* BNR){
using namespace std;
int sBNR;                          // size of BigNum
ippsGetSize_BN(BNR,&sBNR);         // getting size
IppsBigNumSGN sgn;                 // sign of BigNum
Ipp32u* dBNR=new Ipp32u[sBNR];     // space for BNR data
ippsGet_BN(&sgn,&sBNR,dBNR,BNR);  // getting sign and data
int size=sBNR;
IppsBigNumState* BN=nBN(size,dBNR);// neglecting  sign
Ipp8u* vBN = new Ipp8u [size*4];   // which is typed below
ippsGetOctString_BN(vBN, size*4, BN);
if(Msg)
  cout << Msg;                      // header
cout.fill('0');
cout << hex ;
if(sgn==0) cout<< "-";              // sign
for(int n=0; n<size*4; n++){
  cout.width(2);
  cout <<(int)vBN[n];               // value
```

```
}
cout << dec;
cout.fill(' ');
cout << endl;
delete [] vBN;
}
```

Bibliography

[ARCFOUR] Kaukonen, K., Thayer, R., *A Stream Cipher Encryption Algorithm "Arcfour"*, December 1999, *(http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt)*.

[BLOWFISH] Schneier, B., *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, December 1993, *(http://www.schneier.com/paper-blowfish-fse.html)*

[FIPS 46-3] *Data Encryption Standard (DES)*, Federal Information Processing Standards Publication 46-3, October 1999, *(http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf)*.

[FIPS 113] *Computer Data Authentication*, Federal Information Processing Standards Publication 113, May 1985, *(http://www.itl.nist.gov/fipspubs/fip113.htm)*.

[FIPS 186] *Digital Signature Standard (DSS)*, Federal Information Processing Standards Publication 186, May 1994, *(http://www.itl.nist.gov/fipspubs/fip186.htm)*.

[FIPS 186-3] *Digital Signature Standard (DSS)*, Federal Information Processing Standards Publication 186-3, June 2009, *(http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf)*.

[FIPS 197] *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197, November 2001, *(http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf)*.

[FIPS 198] *The Keyed-Hash Message Authentication Code (HMAC)*, Federal Information Processing Standards Publication 198, March 2002, *(http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf)*.

[HAC] Menezes, A.J., Oorschot, P.C., Vanstone, S.A., *Handbook of Applied Cryptography*, CRC Press, 2001,
(http://www.cacr.math.uwaterloo.ca/hac/index.htm).

[MANUAL] *Intel® Integrated Performance Primitives Reference Manual, Volume 4: Cryptography*, IPP 7.1, Intel, August 2012.
(http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-documentation/).

[PKCS 1] PKCS#1 v2.1: *RSA Cryptography Standard*, RSA Laboratories, June 2002,
(ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf).

[PKCS 3] PKCS#3: *Diffie-Hellman Key-Agreement Standard*, RSA Laboratories, November 1993,
(ftp://ftp.rsasecurity.com/pub/pkcs/ps/pkcs-3.ps).

[RC5] Rivest, R., *The RC5 Encryption Algorithm*, March 1997,
(people.csail.mit.edu/rivest/Rivest-rc5rev.ps).

[RFC 1321] Rivest, R., *The MD5 Message-Digest Algorithm*, RFC 1321, April 1992,
(http://www.faqs.org/rfcs/rfc1321.html).

[RFC 2202] Cheng, P., Glenn R., *Test Cases for HMAC-MD5 and HMAC-SHA-1*, RFC 2202, September 1997,
(http://www.faqs.org/rfcs/rfc2202.html).

[RFC 3566] Frankel, S., Herbert, H., *The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec*, RFC 3566, September 2003,
(http://www.apps.ietf.org/rfc/rfc3566.html).

[RFC 4493] Song, JH., Poovendran, R., Lee, J., Iwata, T., *The AES-CMAC Algorithm*, RFC 4493, June 2006,
(http://www.faqs.org/rfcs/rfc4493.html).

[SEC 1] *Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography*, Certicom Research, May 2009,
(http://www.secg.org/download/aid-780/sec1-v2.pdf).

[SEC 2] *Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters*, Certicom Research, january 2010, *(http://www.secg.org/download/aid-784/sec2-v2.pdf)*.

[SMART] Smart, N., *Cryptography: An Introduction*, McGraw-Hill, 2002, *(http://www.cs.bris.ac.uk/~nigel/Crypto_Book/book.ps)*.

[SP 800-38A] *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*, NIST Special Publication 800-38A, December 2001, *(http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf)*.

[SP 800-67] *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*, NIST Special Publication 800-67, Version 1.1, May 2008, *(http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf)*.

[TWOFISH] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N., *Twofish: A 128-Bit Block Cipher*, June 1998, *(http://www.schneier.com/paper-twofish-paper.pdf)*.