The AMASTRID Block Ciphering Method (Versions 1.0.0, 1.1.0) - Design, Security Considerations, and Comments to C++ Reference Code

by George Anescu, January 2011
george.anescu@sc-gen.com, www.sc-gen.com

1. Introduction

AMASTRID 1.0.0 is dynamic size block ciphering method, i.e. it can encrypt/decrypt blocks of data of any size (at word level) and the block size is determined dynamically by an associated Pseudo-Random Number Generator (PRNG). Any good PRNG can be used, like the ones used in the cryptographic stream ciphering methods, but my OCELOT stream ciphering method is preferred because both methods can be initialized simulataneously by applying the same key expansion method. A dynamic size block ciphering method is adavantageous compared to a fixed block size ciphering method because it eliminates the security weaknesses related to the use of the ECB mode. In this way the decryption of the current block is synchronous (not dependent on any of the previous data blocks, like fore example in the case of CBC and PCBC modes), which makes the method more robust in noisy data transmissions. A dynamic size block ciphering method can be used like a stream ciphering method, but without the need to change the encryption key for each encryption/decryption operation.
AMASTRID 1.1.0 is a fixed block size version of the method implemanted mainly for benchmark tests to see how AMASTRID compares from the efficiency perspective with other well known block ciphering methods.

1.1 Design Goals and Characteristics

Simplicity and Mathematical elegance;

Scalability - the capability to work with data blocks of any size (AMASTRID version 1.0.0). The size granularity is at word level, but versions working at byte level can be easily implemented.

Extensibility - the simple XOR and arithmetic modulo operations at byte or word level can be naturally extended for increased security to latin squares (also known as quasigroups) operations having the dimensions 256×256;

Variable key data size - the initial key data is expanded in a large key space. Any practical initial key data size can be used, the larger the key data size the better being the security provided against brute force attacks;

Security versus performance trade-offs - based on a parameterized number of rounds in the primitive block ciphering function;

Resistence - to clasical and more modern security attacks.

1.2 Limitations

Efficiency - the method needs to be extensively tested against other well known block ciphering methods. For this purpose the more efficient version AMASTRID 1.1.0 was created. I am not aware of any other existing dynamic size block ciphering method for testing AMASTRID 1.1.0 against it. Also the initial key expansion phase could be considered somewhat detrimental to the method's efficiency especially when encrypting/decrypting small amounts of data.

Security Holes - although I designed everything having security in mind, the method is new, never before published and not extensively studied;

2. Design Considerations

2.1 Dynamic Input Block Size

The block size can take values in a range with the maximum value being a power of 2. If we denote by max the maximum value then the minimum value is $max/2 + 1$. The max value is given as a parameter to the class constructor and its possible values in bytes are: $16, 32, 64, 128, 256, 512, 1024$ (or in words $4, 8, 16, 32, 64, 128, 256$), which cover all the present practical needs. For example if max is 128 then the block size can take values between 65 and 128, and if max is 256 then the block size can take values between 129 and 256. The block size is determined in a pseudo-random manner, using the associated PRNG. In the reference code I used my OCELOT PRNG, which has the advantage that it can be initialized very efficientrly by using data from the same keys as AMASTRID. If we use the notation $min = max/2$, then the block size is determined as it is shown in the following code block from the C++ reference implementation (methods $Encrypt()$, $Decrypt()$ and $EncryptFile()$, $DecryptFile()$)

```
_rnd.GetNextByte(rndb);
blocksize = rndb;
// get a random number betwen min+1 and max
blocksize &= (BYTE)min1;
blocksize |= (BYTE)min;
++blocksize;
```

where $min1$ is calculated as $min - 1$. The exceptions to the rule are the cases when the plaintext/ciphertext length is $< max$ and the last 2 blocks when the plaintext/ciphertext has length $> max$. If the message length is $< max$ then it is padded to the block size max. The padding done with pseudo-random data generated by the associated PRNG.

For the last 2 blocks the algorithm detects when the length of the remaining message data becomes $\leq 2max$. The second to last block size is calculated as $remaining\_length/2$ and the last block size is calculated as $remaining\_length - (remaining\_length/2)$. In this way the algorithm avoids the need of any padding for message lengths larger than max.

2.2 Help Functions

Some help functions are used in the primitive hash function and also in the key expansion process:

```
UINT KK1(UINT const& val);
UINT KK2(UINT const& val);
BYTE H1(UINT const& word);
BYTE H2(UINT const& word);
UINT F1(UINT const& val);
UINT F2(UINT const& val);
UINT G1(UINT const& val);
UINT G2(UINT const& val);
void Swap(UINT const& val1, UINT const& val2);
```

The function $KK1()$ is applying the $\_kk1$ permutation to the bytes of the word argument;

The function $KK2()$ is applying the $\_kk2$ array to the bytes of the word argument;

$H1()$ and $H2()$ are two simple hash functions for processing a word value to a byte value;

$F1()$ and $F2()$ are two simple functions for flipping half of the bit positions in a word;

$G1()$ and $G2()$ are two simple shift with rotation functions. $G1()$ shifts a word left 5 positions with rotation. $G2()$ shifts a word right 5 positions with rotation;

The function $Swap()$ is swapping the positions of the $\_kk1$ permutation based on the two word arguments. It is used during the expansion process in the class $Expansion$;

There are also some other help methods in the AMASTRID class, like the methods for transforming bytes and words data in an endianess specific manner (littel endian or big endian). All of them can be easily identified in the source code.

2.3 Initialization

The purpose of the initialization process is to generate the method's keys based on the initial key data and to use the generated key data for initializing the associated OCELOT PRNG. The keys consist in the byte arrays $\_kk1[256]$ and $\_kk2[256]$. $\_kk2$ is generated by expanding the initial key data, while $\_kk1$ is obtained during the same expansion process by swapping the values of an initial permutation array. The swapping applied to the $\_kk1$ permutation array is implemented according to the known Durstenfeld shuffle algorithm, which ensure that the initial $\_kk1$ permutation array is significantly changed by the expansion process. Also, as part of the initialization process, for increased security, the final $\_kk1$ permutation array is applied to the array of prime numbers $\_sarrprimes[256]$ (see method $PrimesPermut()$). In the C++ reference code the initialization process is implemented by method $Initialize()$, which is using the class $Expansion$.

2.4 Primitive Block Encryption/Decryption Functions

The primitive block encryption/decryption functions are implemented in the C++ reference code by methods:

```
// Encrypt Primitive
void Encrypt(UINT* ar, UINT const& len);

// Decrypt Primitive
void Decrypt(UINT* ar, UINT const& len);
```

The data block is spanned iteratively on its full length a number of times determined by the value of $\_iter$ instance variable, which is initialized in the constructor. By means of the $\_iter$ instance variable we can control the trade-off between security and efficiency. Larger values of $\_iter$ increase the security, but decrease the efficiency. A typical value of $\_iter$ is 3.

At each loop iteration 4 positions (indexes) in the data block are involved, maintained in the C++ reference code by the variables $pos1$, $pos2$, $pos1o$, $pos2o$, where $pos1o$ and $pos2o$ are the old values of $pos1$, $pos2$ (at the previous loop iteration). The values in positions $pos1$, $pos2$ are changed in a reversible manner by using the values in the 4 positions and by applying the keys in a data dependent manner, i.e. by generating indexes in the key arrays based on the current processed data. There are multiple cases analyzed depending on the values of the positions $pos1$, $pos2$, $pos1o$, $pos2o$ in order to ensure that the data is always processed in a reversible manner, for example the data is processed differently in the cases $pos1 == pos2$, $pos1 == pos2o$, $pos2 == pos1o$. Before starting each loop processing the initial index positions and the increments are determined in a pseudo-random manner based on parts of the key data. In this way the encryption/decryption path is made key dependent. The increments of the indexes $pos1$ and $pos2$, $inc1$ and $inc2$, are determined in a pseudo-random manner such that to ensure that all the index positions are covered during a spanning loop. The necessary and sufficient condition in order to obtain this full coverage property is that the index value be a positive number prime with the data block size. For this purpose we use an array $\_sarrprimes[256]$ containing the first 256 prime numbers greater

than 256 (it is because no message block can be larger than 256 words in the current implementation). Any block size is prime with any of the prime numbers in array $\_sarrprimes$, because any block size is less than any of the prime numbers in array $\_sarrprimes$.

Technique: If $size$ is the block size, and we select in a pseudo-random manner (based on key data) 2 prime numbers $prime_1$ and $prime_2$ from array $\_sarrprimes$, then it can be easily proved mathematically that the increments $inc_1 = (prime_1 \mod size)$ and $incr2 = (prime_2 \mod size)$ are also prime with $size$:

Proof: We can consider by reduction to absurdity that the statement is not true, and that there exists a positive divisor $d > 1$ of both inc_1 and $size$. But we have $prime_1 = q_1 size + inc_1$ for some positive integer q_1, and from it we get that d should also divide $prime_1$, which is a contradiction, because by hypothesis $prime_1$ is a prime number, q.e.d.

By calculating the increments inc_1 and inc_2 using the presented technique we can be sure that we obtain the full coverage property.

2.5 Some specifics of Version 1.1.0

All the aspects presented so far apply to Version 1.0.0. The purpose of version Version 1.1.0 is to improve the efficiency as much as possible while preserving the security characteristics of the method. In this way Version 1.1.0 can better compete with other well known tested and benchmarked block ciphering methods. The following main changes apply to Version 1.1.0:

1) The block size is fixed during the hashing and a power of 2.
2) There is a padding involved for the last block. The last block is filled to the block size with data generated by processing the key data (see methods $Encrypt()$ and $EncryptFile()$).
3) An initial vector (IV) is XOR-ed to the first block in the CBC mode of operation. It is generated by method $GenerateIV()$ based on keydata.
4) All the division modulo operations were eliminated, since they were considered too expensive from the efficiency perspective. The two indexes that were determined in a pseudo-random manner based on the array of prime numbers $\_sarrprimes[256]$, as it was explained in 2.4, are now still determined in a pseudo-random manner as odd numbers, since all odd numbers less than the block size posses the full coverage property for block sizes which are powers of 2.

3. Security Considerations

3.1 Brute Force Attack

As it was already explained, the keys consist in the byte arrays $\_kk1[256]$ and $\_kk2[256]$. $\_kk2$ is generated by expanding the initial key data, while $\_kk1$ is obtained during the same expansion process by permuting an initial permutation array. The dimension of the key space for the array $\_kk2$ is 2^{2048} while for the $\_kk1$ permutation array the dimension of the key space is 256!, which can be approximated to 10×2^{1680} by using the well known Stirling's formula $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$. It gives a total key space of $\approx 10 \times 2^{3728}$. The brute force attack on the expanded keys is impossible with the computing capabilities of present day computers, and it will not be possible for long time from now. Such an attack is possible only on the initial key data (before expansion) if it is not carefully generated. A key data size of at least 16 bytes is considered safe at present, but increased key data size can be accomodated as needed without any changes in the code.

3.2 Linear and Differential Cryptanalysis

These types of attacks attempt to decrease the key space in order to make it manageable by the brute force attack. They cannot be successful against AMSTRID for the following reasons:

1) The key space is very large so that the most of it needs to be derived from linear and differential attacks in order to reduce the key space to a size manageable by the brute force attack, which is unusual (usually only a small portion of the key space is derived from linear and differential attacks).
2) The encryption path is key dependent and so it is a part of the secret (more exactly it is dependent on parts of the keys which are unknown not only as data contents, but also as positions), while the key scheduling algorithm is strongly data dependent. These two properties make impossible to derive portions of the keys by applying linear and differential attacks.

4.0 Apendix

4.1 Test Samples, Version 1.0.0

1) iterations=3, max blocksize=16,
key="aaaa", data="aaaaaaaaaax"
hexresult="CA5E0F2D78ADD58B8B261EC61FB48603"

2) iterations=3, max blocksize=16,
key="aaaaaaaaaaaaaaaa", data="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaax"
hexresult="472951582A67B95357D11306A97E1C9370C04A9DD63F68358251CED243CA6BD82C581460"

3) iterations=3, max blocksize=32,
key="aaaaaaaaaaaaaaaa", data="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaxxbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbyyaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaazz"
hexresult="051623C4D1660B7AC4485E9CC71F6C4A33CA501D745A2DACB5F8B43F14175D12F1B860
CFA3E9CA8C8F5FBB5D21C994D576D2737546932DBC5BCA4913AF910B6A589A231840FC23B790F4DC
A223AF95E17E56B498700C76F7A45B2EC447076771"

4.2 C++ Reference Code, Version 1.0.0

```
// Amastrid.h
#ifndef __AMASTRID_H__
#define __AMASTRID_H__

#include "Ocelot1.h"

#include <stdexcept>
#include <cstring>
#include <string>
#include <fstream>

using namespace std;

#define BYTE unsigned char
#define UINT unsigned int
#define ULONG unsigned long long

class Amastrid
{
//*************************************************************************
//
// The AMASTRID block ciphering method, Version 1.0.0 (19 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
```

```
// All right reserved.
//
// This is the C++ implementation of a new block ciphering method called AMASTRID.
// It works with keys of any practical size and with dynamic block size.
//
// COPYRIGHT PROTECTION: The AMASTRID block ciphering method is still under development and
// testing and for this reason the code is freely distributed only for TESTING AND RESEARCH
// PURPOSES. The author is reserving for himself the rights to MODIFY AND MAINTAIN the code,
//  but any ideas about improving the code are welcomed and will be recognized if implemented.
//
// If you are interested in testing the code, in research collaborations for possible security
// holes in the method, or in any other information please contact the author at
// <george.anescu@sc-gen.com>.
//
//*************************************************************************
public:
    // Sizes in bytes
    enum BlockSize
    {
        BlockSize16 = 16, BlockSize32 = 32, BlockSize64 = 64,
        BlockSize128 = 128, BlockSize256 = 256, BlockSize512 = 512, BlockSize1024 = 1024,
    };

    // Constructors
    Amastrid(BlockSize bs4 = BlockSize16, UINT const& iter=3) : _size4(bs4), _iter(iter)
    {
        if (Amastrid::IsBigEndian())
        {
            Bytes2Word = Amastrid::Bytes2WordBE;
            Word2Bytes = Amastrid::Word2BytesBE;
        }
        else
        {
            Bytes2Word = Amastrid::Bytes2WordLE;
            Word2Bytes = Amastrid::Word2BytesLE;
        }
        _size = _size4 >> 2; //size in words
        _size1 = _size - 1;
        // defaults
        memcpy(_kk1, _skk1, 256);
        memcpy(_kk2, _skk2, 256);
    }

    Amastrid(BYTE const* keydata, UINT const& keysize, BlockSize bs4 = BlockSize16,
        int const& iter=3) : _size4(bs4), _iter(iter)
    {
        if (Amastrid::IsBigEndian())
        {
            Bytes2Word = Amastrid::Bytes2WordBE;
            Word2Bytes = Amastrid::Word2BytesBE;
        }
        else
```

```
        {
            Bytes2Word = Amastrid::Bytes2WordLE;
            Word2Bytes = Amastrid::Word2BytesLE;
        }
        _size = _size4 >> 2; // size in words
        _size1 = _size - 1;
        Initialize(keydata, keysize);
        PrimesPermut();
    }

    void Initialize(BYTE const* keydata, UINT const& keysize);

    void Reset()
    {
        _rnd.Reset();
    }

    // Encrypt Primitive
    void Encrypt(UINT* ar, UINT const& len);

    // Decrypt Primitive
    void Decrypt(UINT* ar, UINT const& len);

    // Variable block size encryption
    void Encrypt(BYTE const* data, BYTE* res, ULONG const& length);

    // Variable block size decryption
    void Decrypt(BYTE const* data, BYTE* res, ULONG const& length);

    // Encrypt File
    void EncryptFile(string const& infilepath, string const& outfilepath);

    // Decrypt File
    void DecryptFile(string const& infilepath, string const& outfilepath, ULONG const& length);

private:
    UINT KK1(UINT const& val)
    {
        return _kk1[(BYTE)val] | _kk1[(BYTE)(val>>8)]<<8 | _kk1[(BYTE)(val>>16)]<<16 |
            _kk1[(BYTE)(val>>24)]<<24;
    }

    UINT KK2(UINT const& val)
    {
        return _kk2[(BYTE)val] | _kk2[(BYTE)(val>>8)]<<8 | _kk2[(BYTE)(val>>16)]<<16 |
            _kk2[(BYTE)(val>>24)]<<24;
    }

    void PrimesPermut()
    {
        // Apply the permutation to the array of primes, so that to make it strongly
        // dependent on the key
```

```
    for (int i = 0; i < 256; i++)
    {
        _ap[i] = _sarrprimes[_kk1[i]];
    }
}

static bool IsBigEndian()
{
    static UINT ui = 1;
    // Executed only at first call
    static bool result(reinterpret_cast<BYTE*>(&ui)[0] == 0);
    return result;
}

static UINT Bytes2WordLE(BYTE const* bytes)
{
    return (UINT)(*(bytes+3)) | (UINT)(*(bytes+2)<<8) |
        (UINT)(*(bytes+1)<<16) | (UINT)(*bytes<<24);
}

static UINT Bytes2WordBE(BYTE const* bytes)
{
    return *(UINT*)bytes;
}

static void Word2BytesLE(UINT word, BYTE* bytes)
{
    bytes += 3;
    *bytes = (BYTE)word;
    *(--bytes) = (BYTE)(word>>8);
    *(--bytes) = (BYTE)(word>>16);
    *(--bytes) = (BYTE)(word>>24);
}

static void Word2BytesBE(UINT word, BYTE* bytes)
{
    *bytes = (BYTE)word;
    *(++bytes) = (BYTE)(word>>8);
    *(++bytes) = (BYTE)(word>>16);
    *(++bytes) = (BYTE)(word>>24);
}

// F1 function
static UINT F1(UINT const& val)
{
    return (val & 0x55AA55AA) | (~val & 0xAA55AA55);
}

// F2 function
static UINT F2(UINT const& val)
{
    return (val & 0xAA55AA55) | (~val & 0x55AA55AA);
```

```
}

// G1 function - shift left rotating
static UINT G1(UINT const& val)
{
    // take last 5 bits, shift left 5 positions and make last 5 bits first
    return (val << 5) | ((val & 0xF8000000) >> 27);
}

// G2 function - shift right rotating
static UINT G2(UINT const& val)
{
    // Take first 5 bits, shift right 5 positions and make first 5 bits last
    return (val >> 5) | ((val & 0x000001F) << 27);
}

static BYTE H1(UINT const& word)
{
    return (word ^ (word>>24)) + ((word>>16) ^ (word>>8));
}

static BYTE H2(UINT const& word)
{
    return (word + (word>>16)) ^ ((word>>8) + (word>>24));
}

// Input File Length
static ULONG FileLength(ifstream& in);

static void ShiftLeftRot(UINT* ar, UINT len, UINT shift)
{
    UINT shift4 = shift << 2;
    UINT ls4 = (len - shift) << 2;
    memcpy(_temp, ar, shift4);
    memmove(ar, (BYTE*)ar+shift4, ls4);
    memcpy((BYTE*)ar+ls4, _temp, shift4);
}

static void ShiftRightRot(UINT* ar, UINT len, UINT shift)
{
    UINT shift4 = shift << 2;
    UINT ls4 = (len - shift) << 2;
    memcpy(_temp, (BYTE*)ar+ls4, shift4);
    memmove((BYTE*)ar+shift4, ar, ls4);
    memcpy(ar, _temp, shift4);
}

void Bytes2Words(UINT* ar1, UINT* ar2, int const& len)
{
    for(register int i=0; i<len; i++)
    {
        ar2[i] = Bytes2Word((BYTE*)&ar1[i]);
```

```
        }
    }

    void Words2Bytes(UINT* ar1, BYTE* ar2, int const& len)
    {
        BYTE* pbytes = ar2;
        for(register int i=0; i<len; i++,pbytes+=4)
        {
            Word2Bytes(ar1[i], pbytes);
        }
    }

    UINT (*Bytes2Word)(BYTE const* bytes);
    void (*Word2Bytes)(UINT word, BYTE* bytes);

    const static BYTE _skk1[256];
    const static BYTE _skk2[256];
    const static short _sarrprimes[256];

    static UINT _temp[256];

    Ocelot1 _rnd;
    BYTE _kk1[256];
    BYTE _kk2[256];
    short _ap[256];

    UINT _size4; // size in bytes
    UINT _size;  // size in words
    UINT _size1;
    UINT _iter;
};

#endif // __AMASTRID_H__

// Amastrid.cpp
#include "Amastrid.h"
#include "Expansion.h"

//***************************************************************************
//
// The Amastrid block ciphering method, Version 1.0.0 (19 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
//***************************************************************************

const BYTE Amastrid::_skk1[256] = {
    99, 23, 136, 230, 240, 243, 115, 247, 6, 13, 48, 154, 119,
    174, 93, 185, 159, 221, 8, 67, 129, 189, 62, 222, 31, 152,
    139, 82, 101, 65, 109, 211, 187, 186, 151, 76, 0, 116, 103,
    7, 11, 34, 144, 209, 80, 137, 236, 232, 94, 225, 21, 194,
    254, 60, 179, 14, 17, 46, 125, 4, 251, 85, 18, 155, 206, 104,
```

```
    195, 237, 56, 22, 181, 73, 63, 213, 38,  3, 203, 89, 190,
    123, 239, 193, 164, 51, 1, 150, 19, 147, 92, 72, 35, 117,
    173, 50, 15, 248, 79, 69, 217, 16, 227, 9, 39, 96, 77, 30, 64,
    57, 255, 78, 169, 199, 54, 111, 176, 242, 58, 106, 5, 220, 45,
    107, 235, 105, 145, 202, 229, 128, 29, 118, 100, 74, 163,
    83, 24, 182, 216, 200, 12, 26, 98, 146, 110, 201, 126, 20, 86,
    143, 55, 175, 167, 233, 166, 122, 158, 183, 204, 141, 215,
    68, 142, 241, 2, 124, 102, 52, 191, 218, 244, 49, 28, 59, 36,
    178, 131, 130, 245, 42, 196, 10, 250, 197, 112, 228, 135,
    168, 108, 75, 40, 210, 44, 88, 160, 171, 223, 249, 91, 43,
    165, 188, 97, 95, 25, 138, 231, 192, 113, 87, 214, 47, 32,
    207, 226, 71, 81, 53, 61, 90, 180, 132, 114, 127, 156, 234,
    133, 120, 41, 27, 121, 148, 238, 184, 212, 149, 66, 134, 246,
    33, 205, 162, 84, 198, 37, 161, 153, 157, 172, 252, 208, 140,
    170, 219, 224, 70, 253, 177,
};

const BYTE Amastrid::_skk2[256] = {
    198, 166, 58, 28, 230, 148, 233, 35, 6, 113, 232, 150, 42, 186,
    92, 103, 50, 171, 188, 14, 246, 139, 201, 202, 9, 217, 193,
    253, 56, 130, 154, 165, 26, 254, 170, 65, 138, 80, 95, 43, 229,
    168, 143, 39, 66, 108, 102, 99, 153, 116, 192, 91, 234,
    238, 132, 12, 97, 177, 209, 55, 207, 23, 68, 86, 62, 147, 3,
    163, 199, 178, 18, 181, 105, 69, 29, 160, 211, 1, 22, 60, 15,
    8, 204, 224, 180, 34, 237, 54, 226, 115, 32, 222, 247, 25, 145,
    81, 183, 21, 94, 44, 215, 109, 175, 243, 106, 210, 31, 83,
    127, 197, 96, 104, 218, 189, 67, 212, 219, 158, 98, 241, 100,
    37, 38, 195, 200, 244, 242, 76, 47, 4, 214, 78, 20, 162, 164,
    252, 190, 194, 134, 112, 129, 172, 16, 72, 221, 119, 52, 84,
    10, 121, 33, 51, 40, 5, 36, 213, 151, 74, 87, 64, 191, 250,
    120, 85, 140, 61, 245, 2, 135, 235, 30, 90, 220, 88, 146, 79,
    227, 137, 216, 48, 123, 208, 110, 73, 184, 239, 159, 53, 71,
    17, 27, 93, 136, 251, 77, 144, 223, 11, 182, 57, 107, 185, 75,
    63, 187, 205, 59, 169, 142, 13, 255, 114, 82, 248, 149, 176,
    89, 70, 167, 173, 152, 133, 122, 126, 101, 249, 131, 49, 0, 228,
    19, 111, 45, 161, 41, 128, 124, 196, 203, 117, 206, 141,
    157, 179, 118, 125, 240, 24, 236, 7, 174, 231, 155, 156, 46, 225,
};

const short Amastrid::_sarrprimes[256] = {
    719, 1229, 1759, 467, 971, 1489, 2053, 709, 1223, 1753,
    463, 967, 1487, 2039, 701, 1217, 1747, 461, 953, 1483,
    2029, 691, 1213, 1741, 457, 947, 1481, 2027, 683, 1201,
    1733, 449, 941, 1471, 2017, 677, 1193, 1723, 443, 937,
    1459, 2011, 673, 1187, 1721, 439, 929, 1453, 2003, 661,
    1181, 1709, 433, 919, 1451, 1999, 659, 1171, 1699, 431,
    911, 1447, 1997, 653, 1163, 1697, 421, 907, 1439, 1993,
    647, 1153, 1693, 419, 887, 1433, 1987, 643, 1151, 1669,
    409, 883, 1429, 1979, 641, 1129, 1667, 401, 881, 1427,
    1973, 631, 1123, 1663, 397, 877, 1423, 1951, 619, 1117,
    1657, 389, 863, 1409, 1949, 617, 1109, 1637, 383, 859,
    1399, 1933, 613, 1103, 1627, 379, 857, 1381, 1931, 607,
```

```
    1097, 1621, 373, 853, 1373, 1913, 601, 1093, 1619, 367,
    839, 1367, 1907, 599, 1091, 1613, 359, 829, 1361, 1901,
    593, 1087, 1609, 353, 827, 1327, 1889, 587, 1069, 1607,
    349, 823, 1321, 1879, 577, 1063, 1601, 347, 821, 1319,
    1877, 571, 1061, 1597, 337, 811, 1307, 1873, 569, 1051,
    1583, 331, 809, 1303, 1871, 563, 1049, 1579, 317, 797,
    1301, 1867, 557, 1039, 1571, 313, 787, 1297, 1861, 547,
    1033, 1567, 311, 773, 1291, 1847, 541, 1031, 1559, 307,
    769, 1289, 1831, 523, 1021, 1553, 293, 761, 1283, 1823,
    521, 1019, 1549, 283, 757, 1279, 1811, 509, 1013, 1543,
    281, 751, 1277, 1801, 503, 1009, 1531, 277, 743, 1259,
    1789, 499, 997, 1523, 271, 739, 1249, 1787, 491, 991,
    1511, 269, 733, 1237, 1783, 487, 983, 1499, 263, 727,
    1231, 1777, 479, 977, 1493, 257,
};

UINT Amastrid::_temp[256];

void Amastrid::Initialize(BYTE const* keydata, UINT const& keysize)
{
    UINT kwords[64];
    int keysizew = keysize >> 2;
    if (keysize & 3) keysizew++;
    kwords[keysizew-1] = 0;
    memcpy(kwords, keydata, keysize);
    Bytes2Words(kwords, kwords, keysizew);
    Expansion exp;
    exp.Expand(kwords, keysizew, _kk2);
    exp.GetKK1(_kk1);
    // _kk2 in words
    memcpy(kwords, _kk2, 256);
    Bytes2Words(kwords, kwords, 64);
    _rnd.Initialize(Ocelot1::OCELOTSize64 , kwords, _kk1);
    PrimesPermut();
}

void Amastrid::Encrypt(UINT* ar, UINT const& len)
{
    UINT len1 = len - 1;
    UINT c1, c2, c1o, c2o;
    int kpos1, kinc1, kpos2, kinc2;
    int pos1, pos2, pos1o, pos2o, inc1, inc2;
    BYTE temp[4];
    temp[0] = _kk1[(BYTE)len]; temp[1] = _kk2[temp[0]];
    temp[2] = _kk1[temp[1]]; temp[3] = _kk2[temp[2]];
    UINT kk1 = KK2(Bytes2Word(temp));
    temp[0] = _kk2[(BYTE)len]; temp[1] = _kk1[temp[0]];
    temp[2] = _kk2[temp[1]]; temp[3] = _kk1[temp[2]];
    UINT kk2 = KK1(Bytes2Word(temp));
    int i, k;
    for (k = 0; k < (int)_iter; k++)
    {
```

```
kpos1 = KK1(kk1);
kinc1 = Amastrid::F1(KK2(kk2));
kpos2 = KK2(kk1);
kinc2 = KK1(Amastrid::F2(kk2));
kk1 = KK2(kpos1);
kk2 = KK1(kinc1);
inc1 = _ap[H1(kinc1)] % len;
inc2 = _ap[H2(kinc2)] % len;
pos1 = kpos1 % len;
if (k > 0)
{
    if (pos1 != pos2)
    {
        ar[pos1] = Amastrid::F1(ar[pos1]);
        ar[pos2] = Amastrid::G1(ar[pos2]);
        if (pos1 > pos2)
        {
            Amastrid::ShiftRightRot(ar, len, pos1 - pos2);
        }
        else
        {
            Amastrid::ShiftLeftRot(ar, len, pos2 - pos1);
        }
    }
}
pos2 = kpos2 % len;
if ((inc1 == inc2) && (pos1 == pos2))
{
    pos2++;
    if (pos2 >= (int)len) pos2 = 0;
}
pos1o = -1;
pos2o = -1;
c1o = kk1;
c2o = kk2;
for (i = 0; i < (int)len; i++)
{
    c1 = ar[pos1];
    c2 = ar[pos2];
    if (pos1 == pos2)
    {
        c1 ^= KK1(c1o);
        c1 += Amastrid::F1(c2o);
        c1 ^= Amastrid::F2(KK2(c2o));
        c1 += Amastrid::G1(c1o);
        c2 = c1;
    }
    else if (pos1 == pos2o)
    {
        if (pos2 == pos1o)
        {
            c1 += Amastrid::F2(c2);
```

```
                    c1 ^= KK2(c2);
                    c2 ^= Amastrid::F1(c1);
                    c2 += Amastrid::G2(KK1(c1));
                }
                else
                {
                    c1 ^= Amastrid::F1(KK2(c1o));
                    c1 += Amastrid::F2(c2);
                    c2 ^= KK1(c1);
                    c2 += Amastrid::G2(c1o);
                }
            }
            else if (pos2 == pos1o)
            {
                c1 += KK1(c2);
                c1 ^= Amastrid::G2(c2o);
                c2 += Amastrid::F1(KK2(c2o));
                c2 ^= Amastrid::F2(c1);
            }
            else
            {
                c1 ^= KK1(c1o);
                c1 += Amastrid::F2(c2);
                c1 ^= Amastrid::G1(c2o);
                c2 += KK2(c2o);
                c2 ^= Amastrid::F1(c1);
                c2 += c1o;
            }
            ar[pos1] = c1;
            ar[pos2] = c2;
            if (i < (int)len1)
            {
                c1o = c1;
                c2o = c2;
                pos1o = pos1;
                pos2o = pos2;
                pos1 += inc1;
                if (pos1 >= (int)len) pos1 -= len;
                pos2 += inc2;
                if (pos2 >= (int)len) pos2 -= len;
            }
        }
    }
}

void Amastrid::Decrypt(UINT* ar, UINT const& len)
{
    static UINT kk1[10];
    static UINT kk2[10];
    static int kpos1[10];
    static int kinc1[10];
    static int kpos2[10];
```

```
static int kinc2[10];
UINT len1 = len - 1;
UINT c1, c2, c1o, c2o;
int pos1, pos2, pos1o, pos2o, pos1temp, inc1, inc2;
BYTE temp[4];
temp[0] = _kk1[(BYTE)len]; temp[1] = _kk2[temp[0]];
temp[2] = _kk1[temp[1]]; temp[3] = _kk2[temp[2]];
UINT var1 = KK2(Bytes2Word(temp));
temp[0] = _kk2[(BYTE)len]; temp[1] = _kk1[temp[0]];
temp[2] = _kk2[temp[1]]; temp[3] = _kk1[temp[2]];
UINT var2 = KK1(Bytes2Word(temp));
int i, k;
for (i = 0; i < (int)_iter; i++)
{
    kpos1[i] = KK1(var1);
    kinc1[i] = Amastrid::F1(KK2(var2));
    kpos2[i] = KK2(var1);
    kinc2[i] = KK1(Amastrid::F2(var2));
    kk1[i] = (var1 = KK2(kpos1[i]));
    kk2[i] = (var2 = KK1(kinc1[i]));
}
for (k = _iter - 1; k >= 0; k--)
{
    inc1 = len - _ap[H1(kinc1[k])] % len;
    inc2 = len - _ap[H2(kinc2[k])] % len;
    if (k < (int)_iter - 1)
    {
        pos1temp = pos1;
    }
    pos1 = kpos1[k] % len;
    pos2 = kpos2[k] % len;
    if ((inc1 == inc2) && (pos1 == pos2))
    {
        pos2++;
        if (pos2 >= (int)len) pos2 = 0;
    }
    pos1 += inc1;
    if (pos1 >= (int)len) pos1 -= len;
    pos2 += inc2;
    if (pos2 >= (int)len) pos2 -= len;
    if (k < (int)_iter - 1)
    {
        if (pos1temp != pos2)
        {
            if (pos1temp > pos2)
            {
                Amastrid::ShiftLeftRot(ar, len, pos1temp - pos2);
            }
            else
            {
                Amastrid::ShiftRightRot(ar, len, pos2 - pos1temp);
            }
```

```
        ar[pos1temp] = Amastrid::F1(ar[pos1temp]);
        ar[pos2] = Amastrid::G2(ar[pos2]);
    }
}
for (i = len1; i >= 0; i--)
{
    if (i > 0)
    {
        pos1o = pos1 + inc1;
        if (pos1o >= (int)len) pos1o -= len;
        pos2o = pos2 + inc2;
        if (pos2o >= (int)len) pos2o -= len;
        c1o = ar[pos1o];
        c2o = ar[pos2o];
    }
    else //i == 0
    {
        c1o = kk1[k];
        c2o = kk2[k];
    }
    c1 = ar[pos1];
    c2 = ar[pos2];
    if (pos1 == pos2)
    {
        c1 -= Amastrid::G1(c1o);
        c1 ^= Amastrid::F2(KK2(c2o));
        c1 -= Amastrid::F1(c2o);
        c1 ^= KK1(c1o);
        c2 = c1;
    }
    else if (pos1 == pos2o)
    {
        if (pos2 == pos1o)
        {
            c2 -= Amastrid::G2(KK1(c1));
            c2 ^= Amastrid::F1(c1);
            c1 ^= KK2(c2);
            c1 -= Amastrid::F2(c2);
        }
        else
        {
            c2 -= Amastrid::G2(c1o);
            c2 ^= KK1(c1);
            c1 -= Amastrid::F2(c2);
            c1 ^= Amastrid::F1(KK2(c1o));
            c2o = c1;
        }
    }
    else if (pos2 == pos1o)
    {
        c2 ^= Amastrid::F2(c1);
        c2 -= Amastrid::F1(KK2(c2o));
```

```
                c1 ^= Amastrid::G2(c2o);
                c1 -= KK1(c2);
                c1o = c2;
            }
            else
            {
                c2 -= c1o;
                c2 ^= Amastrid::F1(c1);
                c2 -= KK2(c2o);
                c1 ^= Amastrid::G1(c2o);
                c1 -= Amastrid::F2(c2);
                c1 ^= KK1(c1o);
            }
            ar[pos1] = c1;
            ar[pos2] = c2;
            if (i > 0)
            {
                pos1 = pos1o;
                pos2 = pos2o;
                c1 = c1o;
                c2 = c2o;
            }
        }
    }
}

//Variable block size encryption
void Amastrid::Encrypt(BYTE const* data, BYTE* res, ULONG const& length)
{
    UINT words[256]; //work buffer
    int max4 = _size4;
    int max = _size;
    UINT rndw;
    if (length <= max4)
    {
        int lenw = (int)(length >> 2);
        if (length & 3)
        {
            words[lenw] = 0;
            lenw++;
        }
        memcpy(words, data, (int)length);
        Bytes2Words(words, words, lenw);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        if (lenw < max)
        {
            // padding
            for (int i = lenw; i<max; i++)
            {
                _rnd.GetNextWord(words[i]);
            }
```

```
    }
    Encrypt(words, max);
    Words2Bytes(words, res, max);
    return;
}
ULONG pos = 0; //current position
int max4_2 = max4 << 1;
int min4 = max4 >> 1;
int min = max >> 1;
int min1 = min - 1;
BYTE rndb;
UINT blocksize, blocksize4;
while (true)
{
    ULONG len = length - pos; //remaining
    if (len <= max4_2)
    {
        // just 2 blocks
        int len1 = (int)(len >> 3);
        if ((len & 7) > 4)
        {
            len1++;
        }
        len1 <<= 2;
        memcpy(words, data+pos, len1);
        int lenw1 = len1 >> 2;
        Bytes2Words(words, words, lenw1);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        Encrypt(words, lenw1);
        Words2Bytes(words, res + pos, lenw1);
        pos += len1;
        len1 = (int)(len - len1);
        lenw1 = (int)(len1 >> 2);
        if (len1 & 3)
        {
            words[lenw1] = 0;
            lenw1++;
        }
        memcpy(words, data+pos, len1);
        Bytes2Words(words, words, lenw1);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        Encrypt(words, lenw1);
        Words2Bytes(words, res + pos, lenw1);
        return;
    }
    else
    {
        _rnd.GetNextByte(rndb);
        blocksize = rndb;
        // get a random number betwen min+1 and max
```

```
            blocksize &= (BYTE)min1;
            blocksize |= (BYTE)min;
            ++blocksize;
            blocksize4 = blocksize << 2;
            memcpy(words, data+pos, blocksize4);
            Bytes2Words(words, words, blocksize);
            _rnd.GetNextWord(rndw);
            words[0] ^= rndw;
            Encrypt(words, blocksize);
            Words2Bytes(words, res + pos, blocksize);
            pos += blocksize4;
        }
    }
}

// Variable block size decryption
void Amastrid::Decrypt(BYTE const* data, BYTE* res, ULONG const& length)
{
    UINT words[BlockSize256]; //work buffer
    int max4 = _size4;
    int max = _size;
    UINT rndw;
    if (length <= max4)
    {
        memcpy(words, data, max4);
        Bytes2Words(words, words, max);
        Decrypt(words, max);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        int lenw = (int)(length >> 2);
        if (length & 3)
        {
            lenw++;
        }
        Words2Bytes(words, res, lenw);
        return;
    }
    ULONG pos = 0; // current position
    int max4_2 = max4 << 1;
    int min4 = max4 >> 1;
    int min = max >> 1;
    int min1 = min - 1;
    BYTE rndb;
    UINT blocksize, blocksize4;
    while (true)
    {
        ULONG len = length - pos; // remaining
        if (len <= max4_2)
        {
            // just 2 blocks
            int len1 = (int)(len >> 3);
            if ((len & 7) > 4)
```

```
            {
                len1++;
            }
            len1 <<= 2;
            memcpy(words, data+pos, len1);
            int lenw1 = len1 >> 2;
            Bytes2Words(words, words, lenw1);
            Decrypt(words, lenw1);
            _rnd.GetNextWord(rndw);
            words[0] ^= rndw;
            Words2Bytes(words, res + pos, lenw1);
            pos += len1;
            len1 = (int)(len - len1);
            lenw1 = (int)(len1 >> 2);
            if (len1 & 3)
            {
                lenw1++;
            }
            len1 = lenw1 << 2;
            memcpy(words, data+pos, len1);
            Bytes2Words(words, words, lenw1);
            Decrypt(words, lenw1);
            _rnd.GetNextWord(rndw);
            words[0] ^= rndw;
            Words2Bytes(words, res + pos, lenw1);
            return;
        }
        else
        {
            _rnd.GetNextByte(rndb);
            blocksize = rndb;
            // get a random number betwen min+1 and max
            blocksize &= (BYTE)min1;
            blocksize |= (BYTE)min;
            ++blocksize;
            blocksize4 = blocksize << 2;
            memcpy(words, data+pos, blocksize4);
            Bytes2Words(words, words, blocksize);
            Decrypt(words, blocksize);
            _rnd.GetNextWord(rndw);
            words[0] ^= rndw;
            Words2Bytes(words, res + pos, blocksize);
            pos += blocksize4;
        }
    }
}

// Input File Length
ULONG Amastrid::FileLength(ifstream& in)
{
    // Check first the file's state
    if(!in.is_open() || in.bad())
```

```
    {
        throw runtime_error("FileLength(), file not opened or in bad state.");
    }
    // Get current position
    streampos currpos = in.tellg();
    // Move to the end
    in.seekg(0, ios::end);
    streampos endpos = in.tellg();
    // Go Back
    in.seekg(currpos, ios::beg);
    return (ULONG)endpos;
}

// Encrypt File
void Amastrid::EncryptFile(string const& infilepath, string const& outfilepath)
{
    if (infilepath.empty() || outfilepath.empty())
    {
        throw runtime_error("Amastrid::EncryptFile(), empty file path.");
    }
    try
    {
        UINT words[BlockSize256]; // work buffer
        int max4 = _size4;
        int max = _size;
        ifstream ifs(infilepath.c_str());
        if(!ifs)
        {
            throw runtime_error("Amastrid::EncryptFile(), cannot open input file.");
        }
        ULONG length = FileLength(ifs);
        ofstream ofs(outfilepath.c_str());
        if(!ofs)
        {
            throw runtime_error("Amastrid::EncryptFile(), cannot open output file.");
        }
        UINT rndw;
        if (length <= max4)
        {
            int lenw = (int)(length >> 2);
            if (length & 3)
            {
                words[lenw] = 0;
                lenw++;
            }
            ifs.read((char*)words, (streamsize)length);
            Bytes2Words(words, words, lenw);
            _rnd.GetNextWord(rndw);
            words[0] ^= rndw;
            if (lenw < max)
            {
                // padding
```

```
        for (int i = lenw; i<max; i++)
        {
            _rnd.GetNextWord(words[i]);
        }
    }
    Encrypt(words, max);
    Words2Bytes(words, (BYTE*)words, max);
    ofs.write((char*)words, max4);
    ifs.close();
    ofs.close();
    return;
}
ULONG pos = 0; // current position
int max4_2 = max4 << 1;
int min4 = max4 >> 1;
int min = max >> 1;
int min1 = min - 1;
BYTE rndb;
UINT blocksize, blocksize4;
while (true)
{
    ULONG len = length - pos; // remaining
    if (len <= max4_2)
    {
        // just 2 blocks
        int len1 = (int)(len >> 3);
        if ((len & 7) > 4)
        {
            len1++;
        }
        len1 <<= 2;
        ifs.read((char*)words, (streamsize)len1);
        int lenw1 = len1 >> 2;
        Bytes2Words(words, words, lenw1);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        Encrypt(words, lenw1);
        Words2Bytes(words, (BYTE*)words, lenw1);
        ofs.write((char*)words, len1);
        pos += len1;
        len1 = (int)(len - len1);
        lenw1 = (int)(len1 >> 2);
        if (len1 & 3)
        {
            words[lenw1] = 0;
            lenw1++;
        }
        ifs.read((char*)words, (streamsize)len1);
        Bytes2Words(words, words, lenw1);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        Encrypt(words, lenw1);
```

```
                Words2Bytes(words, (BYTE*)words, lenw1);
                ofs.write((char*)words, lenw1 << 2);
                ifs.close();
                ofs.close();
                return;
            }
            else
            {
                _rnd.GetNextByte(rndb);
                blocksize = rndb;
                // get a random number betwen min+1 and max
                blocksize &= (BYTE)min1;
                blocksize |= (BYTE)min;
                ++blocksize;
                blocksize4 = blocksize << 2;
                ifs.read((char*)words, (streamsize)blocksize4);
                Bytes2Words(words, words, blocksize);
                _rnd.GetNextWord(rndw);
                words[0] ^= rndw;
                Encrypt(words, blocksize);
                Words2Bytes(words, (BYTE*)words, blocksize);
                ofs.write((char*)words, blocksize4);
                pos += blocksize4;
            }
        }
    }
    catch (exception const& ex)
    {
        throw runtime_error("Amastrid::EncryptFile(), file read error: " +
            string(ex.what()));
    }
}

// Decrypt File
void Amastrid::DecryptFile(string const& infilepath, string const& outfilepath,
    ULONG const& length)
{
    if (infilepath.empty() || outfilepath.empty())
    {
        throw runtime_error("Amastrid::DecryptFile(), empty file path.");
    }
    try
    {
        UINT words[BlockSize256]; // work buffer
        int max4 = _size4;
        int max = _size;
        ifstream ifs(infilepath.c_str());
        if(!ifs)
        {
            throw runtime_error("Amastrid::DecryptFile(), cannot open input file.");
        }
        ofstream ofs(outfilepath.c_str());
```

```
if(!ofs)
{
    throw runtime_error("Amastrid::DecryptFile(), cannot open output file.");
}
UINT rndw;
if (length <= max4)
{
    ifs.read((char*)words, (streamsize)max4);
    Bytes2Words(words, words, max);
    Decrypt(words, max);
    _rnd.GetNextWord(rndw);
    words[0] ^= rndw;
    int lenw = (int)(length >> 2);
    if (length & 3)
    {
        lenw++;
    }
    Words2Bytes(words, (BYTE*)words, lenw);
    ofs.write((char*)words, (int)length);
    ifs.close();
    ofs.close();
    return;
}
ULONG pos = 0; // current position
int max4_2 = max4 << 1;
int min4 = max4 >> 1;
int min = max >> 1;
int min1 = min - 1;
BYTE rndb;
UINT blocksize, blocksize4;
while (true)
{
    ULONG len = length - pos; // remaining
    if (len <= max4_2)
    {
        // just 2 blocks
        int len1 = (int)(len >> 3);
        if ((len & 7) > 4)
        {
            len1++;
        }
        len1 <<= 2;
        int lenw1 = len1 >> 2;
        ifs.read((char*)words, len1);
        Bytes2Words(words, words, lenw1);
        Decrypt(words, lenw1);
        _rnd.GetNextWord(rndw);
        words[0] ^= rndw;
        Words2Bytes(words, (BYTE*)words, lenw1);
        ofs.write((char*)words, len1);
        pos += len1;
        len1 = (int)(len - len1);
```

```
                int len1o = len1;
                lenw1 = (int)(len1 >> 2);
                if (len1 & 3)
                {
                    lenw1++;
                }
                len1 = lenw1 << 2;
                ifs.read((char*)words, len1);
                Bytes2Words(words, words, lenw1);
                Decrypt(words, lenw1);
                _rnd.GetNextWord(rndw);
                words[0] ^= rndw;
                Words2Bytes(words, (BYTE*)words, lenw1);
                ofs.write((char*)words, len1o);
                ifs.close();
                ofs.close();
                return;
            }
            else
            {
                _rnd.GetNextByte(rndb);
                blocksize = rndb;
                // get a random number betwen min+1 and max
                blocksize &= (BYTE)min1;
                blocksize |= (BYTE)min;
                ++blocksize;
                blocksize4 = blocksize << 2;
                ifs.read((char*)words, blocksize4);
                Bytes2Words(words, words, blocksize);
                Decrypt(words, blocksize);
                _rnd.GetNextWord(rndw);
                words[0] ^= rndw;
                Words2Bytes(words, (BYTE*)words, blocksize);
                ofs.write((char*)words, blocksize4);
                pos += blocksize4;
            }
        }
    }
    catch (exception const& ex)
    {
        throw runtime_error("Amastrid::DecryptFile(), file read error: " + string(ex.what()));
    }
}

// Expansion.h
#ifndef __EXPANSION_H__
#define __EXPANSION_H__

#include <cstring>

using namespace std;
```

```
#define BYTE unsigned char
#define UINT unsigned int

class Expansion
{
public:
    //Constructor
    Expansion(UINT const& iter=3) : _iter(iter)
    {
        if (Expansion::IsBigEndian())
        {
            Swap = &Expansion::SwapBE;
            Bytes2Word = Expansion::Bytes2WordBE;
            Word2Bytes = Expansion::Word2BytesBE;
        }
        else
        {
            Swap = &Expansion::SwapLE;
            Bytes2Word = Expansion::Bytes2WordLE;
            Word2Bytes = Expansion::Word2BytesLE;
        }
        memcpy(_kk1, _skk1, 256);
    }

    void Reset()
    {
        memcpy(_kk1, _skk1, 256);
    }

    //Expansion
    void Expand(UINT const* data, UINT const& len, BYTE* res);

    void GetKK1(BYTE* kk1)
    {
        memcpy(kk1, _kk1, 256);
    }

private:
    UINT KK1(UINT const& val)
    {
        return _kk1[(BYTE)val] | _kk1[(BYTE)(val>>8)]<<8 |
            _kk1[(BYTE)(val>>16)]<<16 | _kk1[(BYTE)(val>>24)]<<24;
    }

    void SwapLE(UINT const& val1, UINT const& val2)
    {
        register BYTE *p1, *p2;
        register BYTE temp, v1, v2;
        p1 = (BYTE*)(&val1) + 3;
        p2 = (BYTE*)(&val2) + 3;
        //1
        v1 = *p1; v2 = *p2;
```

```
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
    //2
    v1 = *(--p1); v2 = *(--p2);
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
    //3
    v1 = *(--p1); v2 = *(--p2);
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
    //4
    v1 = *(--p1); v2 = *(--p2);
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
}

void SwapBE(UINT const& val1, UINT const& val2)
{
    register BYTE *p1, *p2;
    register BYTE temp, v1, v2;
    p1 = (BYTE*)&val1;
    p2 = (BYTE*)&val2;
    //1
    v1 = *p1; v2 = *p2;
    //if (v1 == v2) v2++;
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
    //2
    v1 = *(++p1); v2 = *(++p2);
    //if (v1 == v2) v2++;
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
    //3
    v1 = *(++p1); v2 = *(++p2);
    //if (v1 == v2) v2++;
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
    //4
    v1 = *(++p1); v2 = *(++p2);
    //if (v1 == v2) v2++;
    temp = _kk1[v1]; _kk1[v1] = _kk1[v2]; _kk1[v2] = temp;
}

static bool IsBigEndian()
{
    static UINT ui = 1;
    //Executed only at first call
    static bool result(reinterpret_cast<BYTE*>(&ui)[0] == 0);
    return result;
}

static UINT Bytes2WordLE(BYTE const* bytes)
{
    return (UINT)(*(bytes+3)) | (UINT)(*(bytes+2)<<8) |
        (UINT)(*(bytes+1)<<16) | (UINT)(*bytes<<24);
}

static UINT Bytes2WordBE(BYTE const* bytes)
{
```

```
    return *(UINT*)bytes;
}

static void Word2BytesLE(UINT word, BYTE* bytes)
{
    bytes += 3;
    *bytes = (BYTE)word;
    *(--bytes) = (BYTE)(word>>8);
    *(--bytes) = (BYTE)(word>>16);
    *(--bytes) = (BYTE)(word>>24);
}

static void Word2BytesBE(UINT word, BYTE* bytes)
{
    *bytes = (BYTE)word;
    *(++bytes) = (BYTE)(word>>8);
    *(++bytes) = (BYTE)(word>>16);
    *(++bytes) = (BYTE)(word>>24);
}

//F1 function
static UINT F1(UINT const& val)
{
    return (val & 0x55AA55AA) | (~val & 0xAA55AA55);
}

//F2 function
static UINT F2(UINT const& val)
{
    return (val & 0xAA55AA55) | (~val & 0x55AA55AA);
}

//G1 function - shift left rotating
static UINT G1(UINT const& val)
{
    //take last 5 bits, shift left 5 positions and make last 5 bits first
    return (val << 5) | ((val & 0xF8000000) >> 27);
}

//G2 function - shift right rotating
static UINT G2(UINT const& val)
{
    //take first 5 bits, shift right 5 positions and make first 5 bits last
    return (val >> 5) | ((val & 0x000001F) << 27);
}

void Bytes2Words(UINT* ar1, UINT* ar2, int const& len)
{
    for(register int i=0; i<len; i++)
    {
        ar2[i] = Bytes2Word((BYTE*)&ar1[i]);
    }
```

```
    }

    void Words2Bytes(UINT* ar1, BYTE* ar2, int const& len)
    {
        BYTE* pbytes = ar2;
        for(register int i=0; i<len; i++,pbytes+=4)
        {
            Word2Bytes(ar1[i], pbytes);
        }
    }

    void (Expansion::*Swap)(UINT const& val1, UINT const& val2);
    UINT (*Bytes2Word)(BYTE const* bytes);
    void (*Word2Bytes)(UINT word, BYTE* bytes);

    const static BYTE _skk1[256];

    UINT _iter;
    BYTE _kk1[256];
};

#endif // __EXPANSION_H__

// Expansion.cpp
#include "Expansion.h"

const BYTE Expansion::_skk1[256] = {
    217, 40, 106, 12, 2, 114, 98, 19, 41, 147, 220, 97, 194,
    35, 171, 105, 10, 235, 80, 56, 178, 253, 21, 39, 187,
    26, 11, 207, 27, 228, 101, 219, 176, 36, 188, 125, 121,
    45, 49, 237, 202, 109, 133, 5, 70, 108, 65, 195, 138, 72,
    58, 168, 60, 37, 161, 151, 110, 96, 198, 66, 174, 126,
    118, 13, 173, 192, 137, 139, 9, 95, 4, 212, 77, 100, 146,
    191, 50, 25, 59, 117, 233, 0, 34, 99, 208, 243, 71, 90,
    53, 8, 160, 222, 143, 177, 119, 230, 123, 46, 145, 136,
    24, 166, 47, 135, 244, 140, 196, 149, 134, 63, 61, 87,
    29, 214, 193, 6, 130, 184, 42, 190, 242, 129, 52, 206, 33,
    250, 247, 155, 86, 84, 23, 165, 88, 180, 239, 81, 16, 162,
    223, 18, 83, 236, 153, 186, 234, 200, 32, 91, 216,
    115, 132, 43, 218, 215, 107, 3, 248, 251, 44, 51, 211,
    226, 15, 201, 62, 20, 240, 22, 163, 231, 57, 246, 255, 1,
    183, 227, 245, 76, 30, 154, 189, 144, 113, 164, 209, 131,
    104, 122, 156, 142, 152, 224, 54, 67, 124, 78, 127, 94,
    175, 68, 167, 103, 48, 221, 205, 148, 150, 79, 7, 181,
    225, 204, 241, 179, 75, 158, 229, 157, 210, 232, 169, 141,
    102, 128, 249, 238, 213, 120, 111, 64, 170, 93, 85, 82,
    28, 252, 116, 112, 203, 17, 197, 254, 14, 185, 73, 92, 31,
    38, 199, 159, 55, 89, 69, 74, 182, 172,
};

void Expansion::Expand(UINT const* data, UINT const& len, BYTE* res)
{
```

```
UINT words[64];
memcpy(words, data, len);
UINT lend2 = len >> 1;
//Propagate differences
BYTE temp[4];
temp[0] = _kk1[0]; temp[1] = _kk1[64]; temp[2] = _kk1[128]; temp[3] = _kk1[192];
register UINT val1 = Bytes2Word(temp);
temp[0] = _kk1[32]; temp[1] = _kk1[96]; temp[2] = _kk1[160]; temp[3] = _kk1[224];
register UINT val2 = Bytes2Word(temp);
register UINT temp1, temp2;
register UINT k, i, ix = lend2;
for (k = 0; k < _iter; k++)
{
    if (k == 1)
    {
        //Combining with output size for compatibility with the general expansion method
        words[0] ^= KK1(256);
    }
    for (i = 0; i < len; i++, ix++)
    {
        if (ix >= len) ix = 0;
        temp1 = words[i];
        temp2 = words[ix];
        val1 ^= KK1(temp1);
        val1 += temp2;
        val2 ^= Expansion::F1(val1);
        val2 += KK1(temp2);
        val2 ^= temp1;
        words[i] = (temp1 = Expansion::G1(temp1) ^ KK1(val1));
        words[ix] = Expansion::F2(temp2) + KK1(val2);
        (this->*Swap)(temp1, val2);
    }
}
//Expanding
UINT resw[64];
i = 0;
ix = lend2;
int max = 64 - lend2; //assumes lend2 <= 64, which is correct for expansion
for (k = 0; k < 64; k++, i++, ix++)
{
    if (i >= len) i = 0;
    if (ix >= len) ix = 0;
    temp1 = words[i];
    temp2 = words[ix];
    val1 += KK1(temp1);
    val1 ^= temp2;
    val2 += Expansion::G1(val1);
    resw[k] = Expansion::F2(val2) ^ KK1(data[i]);
    val2 ^= KK1(temp2);
    val2 += temp1;
    if ((int)k < max)
    {
```

```
            words[i] = Expansion::G2(temp1) + KK1(val1);
            words[ix] = Expansion::F1(temp2) ^ KK1(val2);
        }
        (this->*Swap)(k, val2);
    }
    //get the result
    Words2Bytes(resw, res, 64);
}
```

4.3 Test Samples, Version 1.1.0

1) iterations=3, max blocksize=16,
key="aaaa", data="aaaaaaaaaax"
hexresult="CA5E0F2D78ADD58B8B261EC61FB48603"
hexresult (ECB)="9FDE905DDE1CF023148B6433A8FA2CE3"
hexresult (CBC)="C7FAB646749B5278D51DE9868D9C773A"

2) iterations=3, max blocksize=16,
key="aaaaaaaaaaaaaaaa", data="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaax"
hexresult (ECB)="56527C13D63E1F59F3B3E199493F2A3156527C13D63E1F5
9F3B3E199493F2A31885AB0AEF342D69E0B314C2778C48CFC"
hexresult (CBC)="85C7D54717CF1DF4D3C5450D9799D163E034A1A90071F1F
7EC57420FD0E2991AC0C9D6F289E3BA1605902F6C9684B2B9"

3) iterations=3, max blocksize=32,
key="aaaaaaaaaaaaaaaa", data="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaxxbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbyyaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaazz"
hexresult (ECB)="4964995B7ACC1AB4CBBD46D6765F25266E52499C0F31FE26794D1E58E
54EF05B54562C474DA38EA0F9314B5AFD43238AB200312D5EF18F449B98095B014F896FC1D
6AB944B6EB3BA68808CB8C60BDE63B5FE760ADF9E15A6991AC3206AD57C5E"
hexresult (CBC)="9CEEEFA623E27563C244C676144776B8F48CB9B75438A461F681C6A50
035798B387E41E41971B66DCC9A2F86A14E0303BB0ED471D76BCCF20C42DA147657BE16A51
714B74712A68CABFE29366947D48E1E372A6AAAE9B3534A36C045F7814943"

Counter Mode Test Samples:
1) blocksize=32, iterations=3, key="XXXXXXX":
hexresult="E268FDD8 E837D34A 9A8EF173 D11E01BC 20825594 E72194E2 69A590D7 2C3AD7E0
CB14A29E AF678E96 E2719947 813AFF27 442F0E87 E255E526 2809FB89 4921A523
4585A838 2644FCCD 2C3B8424 8949402B 14CDC887 9B93576E 59021DB2 C04A4DAB
5A5CD1AD 26239C9F 55AE0BCB 48AE0A6E 0C4AB125 48559BA6 BDCCBDC0 EB1DEC55
0B92B6D1 29E9CCFD EDDB70F0 4FF8F1E6 254CC315 1DC3CEDA 136C45C1 B449D5DA
5DF8D1A1 5870EA6B 90C44683 50AE40D1 27B0C57A EC021328 8875174B 5E487A23
341EDAD9 2E119223 CBE4E411 60F8EC8F AAD4284F 71487A84 C6B7E3C3 03438FD4
6659ABB8 8D039683 71BC8292 7C61A4FE 55A57608 E74AD51C F75C2C24 96D1C41F "...

2) blocksize=32, iterations=3, key="YXXXXXX":
hexresult="B63EF2A9 5F382223 3BA75F5C EC6D7C57 2A91F568 638016EF 7B83C08B 835AC403
7BC34E4A 1869C3E3 EE202C32 64A3D4A5 DE56C85C 21ADFD71 E75D1BC6 19BE7DF8
6A792AF2 A3ABE806 B016E21E 57DC50AE DE686BD4 4D45A879 368F30CD 5CA20C6D
43E2128F 4CAD591B E5E677C2 364CEF4D 16E27BC6 1D7EA3BD 1A69BBB1 0866BEC3 "...

3) blocksize=32, iterations=3, key="\0\0\0\0" :

hexresult=”44BAE2BE 482206E2 ACBCDDFB 05FEFCCA 1E114F6D 547C99B0 1B4DD0C8 7489F922
436788F1 CB7A1C59 F7DB446E EEDB8D94 2B1836AD 9D3FFF01 E88F5346 D51F0750
86F8E4F9 AC5F5147 37F7B828 342F2707 149E3A55 44949BEA 0F1B1126 F8F7EE16
E9B4B3FC 60D87FE2 D9ADB4AF E4E0DA10 23FACBE6 198463D8 43A523A3 855C2D53 ”...

4.4 C++ Reference Code, Version 1.1.0

```
// Amastrid.h
#ifndef __AMASTRID_H__
#define __AMASTRID_H__

#include <stdexcept>
#include <cstring>
#include <string>
#include <fstream>

using namespace std;

#define BYTE unsigned char
#define UINT unsigned int
#define ULONG unsigned long long

class Amastrid
{
//*************************************************************************
//
// The AMASTRID block ciphering method (Fast Version), Version 1.1.0 (19 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
// This is the C++ implementation of a new block ciphering method called AMASTRID.
// Similar to Version 1.0.0, Version 1.1.0 works with keys of any practical size, but for
// increased efficiency it works with fixed block size.
//
// COPYRIGHT PROTECTION: The AMASTRID block ciphering method is still under development and
// testing and for this reason the code is freely distributed only for TESTING AND RESEARCH
// PURPOSES. The author is reserving for himself the rights to MODIFY AND MAINTAIN the code,
// but any ideas about improving the code are welcomed and will be recognized if implemented.
//
// If you are interested in testing the code, in research collaborations for possible security
// holes in the method, or in any other information please contact the author at
// <george.anescu@sc-gen.com>.
//
//*************************************************************************
public:
    //Sizes in bytes
    enum BlockSize
    {
        BlockSize16 = 16, BlockSize32 = 32, BlockSize64 = 64, BlockSize128 = 128,
        BlockSize256 = 256, BlockSize512 = 512, BlockSize1024 = 1024,
    };
```

```
enum EncryptionMode
{
    ECB = 0, CBC = 1,
};

//Constructors
Amastrid(BlockSize bs4 = BlockSize16, UINT const& iter=3) : _size4(bs4), _iter(iter)
{
    if (Amastrid::IsBigEndian())
    {
        Bytes2Word = Amastrid::Bytes2WordBE;
        Word2Bytes = Amastrid::Word2BytesBE;
    }
    else
    {
        Bytes2Word = Amastrid::Bytes2WordLE;
        Word2Bytes = Amastrid::Word2BytesLE;
    }
    _size41 = _size4 - 1;
    _size = _size4 >> 2; //size in words
    _size1 = _size - 1;
    //defaults
    memcpy(_kk1, _skk1, 256);
    memcpy(_kk2, _skk2, 256);
}

Amastrid(BYTE const* keydata, UINT const& keysize, BlockSize bs4 = BlockSize16,
    int const& iter=3) : _size4(bs4), _iter(iter)
{
    if (Amastrid::IsBigEndian())
    {
        Bytes2Word = Amastrid::Bytes2WordBE;
        Word2Bytes = Amastrid::Word2BytesBE;
    }
    else
    {
        Bytes2Word = Amastrid::Bytes2WordLE;
        Word2Bytes = Amastrid::Word2BytesLE;
    }
    _size41 = _size4 - 1;
    _size = _size4 >> 2; //size in words
    _size1 = _size - 1;
    Initialize(keydata, keysize);
}

void Initialize(BYTE const* keydata, UINT const& keysize);

//Generate Initialization Vector
void GenerateIV(UINT* iv);

//Encrypt Primitive
void Encrypt(UINT* ar);
```

```
    //Decrypt Primitive
    void Decrypt(UINT* ar);

    //Variable block size encryption
    void Encrypt(BYTE const* data, BYTE* res, ULONG const& length, int em);

    //Variable block size decryption
    void Decrypt(BYTE const* data, BYTE* res, ULONG const& length, int em);

    //Encrypt File
    void EncryptFile(string const& infilepath, string const& outfilepath, int em);

    //Decrypt File
    void DecryptFile(string const& infilepath, string const& outfilepath,
        ULONG const& length, int em);

private:
    UINT KK1(UINT const& val)
    {
        return _kk1[(BYTE)val] | _kk1[(BYTE)(val>>8)]<<8 |
            _kk1[(BYTE)(val>>16)]<<16 | _kk1[(BYTE)(val>>24)]<<24;
    }

    UINT KK2(UINT const& val)
    {
        return _kk2[(BYTE)val] | _kk2[(BYTE)(val>>8)]<<8 |
            _kk2[(BYTE)(val>>16)]<<16 | _kk2[(BYTE)(val>>24)]<<24;
    }

    static bool IsBigEndian()
    {
        static UINT ui = 1;
        //Executed only at first call
        static bool result(reinterpret_cast<BYTE*>(&ui)[0] == 0);
        return result;
    }

    static UINT Bytes2WordLE(BYTE const* bytes)
    {
        return (UINT)(*(bytes+3)) | (UINT)(*(bytes+2)<<8) |
            (UINT)(*(bytes+1)<<16) | (UINT)(*bytes<<24);
    }

    static UINT Bytes2WordBE(BYTE const* bytes)
    {
        return *(UINT*)bytes;
    }

    static void Word2BytesLE(UINT word, BYTE* bytes)
    {
        bytes += 3;
```

```
    *bytes = (BYTE)word;
    *(--bytes) = (BYTE)(word>>8);
    *(--bytes) = (BYTE)(word>>16);
    *(--bytes) = (BYTE)(word>>24);
}

static void Word2BytesBE(UINT word, BYTE* bytes)
{
    *bytes = (BYTE)word;
    *(++bytes) = (BYTE)(word>>8);
    *(++bytes) = (BYTE)(word>>16);
    *(++bytes) = (BYTE)(word>>24);
}

//F1 function
static UINT F1(UINT const& val)
{
    return (val & 0x55AA55AA) | (~val & 0xAA55AA55);
}

//F2 function
static UINT F2(UINT const& val)
{
    return (val & 0xAA55AA55) | (~val & 0x55AA55AA);
}

// G1 function - shift left rotating
static UINT G1(UINT const& val)
{
    // take last 5 bits, shift left 5 positions and make last 5 bits first
    return (val << 5) | ((val & 0xF8000000) >> 27);
}

// G2 function - shift right rotating
static UINT G2(UINT const& val)
{
    // Take first 5 bits, shift right 5 positions and make first 5 bits last
    return (val >> 5) | ((val & 0x000001F) << 27);
}

static BYTE H1(UINT const& word)
{
    return (word ^ (word>>24)) + ((word>>16) ^ (word>>8));
}

static BYTE H2(UINT const& word)
{
    return (word + (word>>16)) ^ ((word>>8) + (word>>24));
}

//Input File Length
static ULONG FileLength(ifstream& in);
```

```
static void ShiftLeftRot(UINT* ar, UINT len, UINT shift)
{
    UINT shift4 = shift << 2;
    UINT ls4 = (len - shift) << 2;
    memcpy(_temp, ar, shift4);
    memmove(ar, (BYTE*)ar+shift4, ls4);
    memcpy((BYTE*)ar+ls4, _temp, shift4);
}

static void ShiftRightRot(UINT* ar, UINT len, UINT shift)
{
    UINT shift4 = shift << 2;
    UINT ls4 = (len - shift) << 2;
    memcpy(_temp, (BYTE*)ar+ls4, shift4);
    memmove((BYTE*)ar+shift4, ar, ls4);
    memcpy(ar, _temp, shift4);
}

void Bytes2Words(UINT* ar1, UINT* ar2, int const& len)
{
    for(register int i=0; i<len; i++)
    {
        ar2[i] = Bytes2Word((BYTE*)&ar1[i]);
    }
}

void Words2Bytes(UINT* ar1, BYTE* ar2, int const& len)
{
    BYTE* pbytes = ar2;
    for(register int i=0; i<len; i++,pbytes+=4)
    {
        Word2Bytes(ar1[i], pbytes);
    }
}

UINT (*Bytes2Word)(BYTE const* bytes);
void (*Word2Bytes)(UINT word, BYTE* bytes);

const static BYTE _skk1[256];
const static BYTE _skk2[256];

static UINT _temp[256];

BYTE _kk1[256];
BYTE _kk2[256];

UINT _size4; //size in bytes
UINT _size41;
UINT _size; //size in words
UINT _size1;
UINT _iter;
```

```
};

#endif // __AMASTRID_H__

// Amastrid.cpp
#include "Amastrid.h"
#include "Expansion.h"

//***************************************************************************
//
// The Amastrid block ciphering method (Fast Version), Version 1.1.0 (19 December 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
//***************************************************************************

const BYTE Amastrid::_skk1[256] = {
    99, 23, 136, 230, 240, 243, 115, 247, 6, 13, 48, 154, 119,
    174, 93, 185, 159, 221, 8, 67, 129, 189, 62, 222, 31, 152,
    139, 82, 101, 65, 109, 211, 187, 186, 151, 76, 0, 116, 103,
    7, 11, 34, 144, 209, 80, 137, 236, 232, 94, 225, 21, 194,
    254, 60, 179, 14, 17, 46, 125, 4, 251, 85, 18, 155, 206, 104,
    195, 237, 56, 22, 181, 73, 63, 213, 38,  3, 203, 89, 190,
    123, 239, 193, 164, 51, 1, 150, 19, 147, 92, 72, 35, 117,
    173, 50, 15, 248, 79, 69, 217, 16, 227, 9, 39, 96, 77, 30, 64,
    57, 255, 78, 169, 199, 54, 111, 176, 242, 58, 106, 5, 220, 45,
    107, 235, 105, 145, 202, 229, 128, 29, 118, 100, 74, 163,
    83, 24, 182, 216, 200, 12, 26, 98, 146, 110, 201, 126, 20, 86,
    143, 55, 175, 167, 233, 166, 122, 158, 183, 204, 141, 215,
    68, 142, 241, 2, 124, 102, 52, 191, 218, 244, 49, 28, 59, 36,
    178, 131, 130, 245, 42, 196, 10, 250, 197, 112, 228, 135,
    168, 108, 75, 40, 210, 44, 88, 160, 171, 223, 249, 91, 43,
    165, 188, 97, 95, 25, 138, 231, 192, 113, 87, 214, 47, 32,
    207, 226, 71, 81, 53, 61, 90, 180, 132, 114, 127, 156, 234,
    133, 120, 41, 27, 121, 148, 238, 184, 212, 149, 66, 134, 246,
    33, 205, 162, 84, 198, 37, 161, 153, 157, 172, 252, 208, 140,
    170, 219, 224, 70, 253, 177,
};

const BYTE Amastrid::_skk2[256] = {
    198, 166, 58, 28, 230, 148, 233, 35, 6, 113, 232, 150, 42, 186,
    92, 103, 50, 171, 188, 14, 246, 139, 201, 202, 9, 217, 193,
    253, 56, 130, 154, 165, 26, 254, 170, 65, 138, 80, 95, 43, 229,
    168, 143, 39, 66, 108, 102, 99, 153, 116, 192, 91, 234,
    238, 132, 12, 97, 177, 209, 55, 207, 23, 68, 86, 62, 147, 3,
    163, 199, 178, 18, 181, 105, 69, 29, 160, 211, 1, 22, 60, 15,
    8, 204, 224, 180, 34, 237, 54, 226, 115, 32, 222, 247, 25, 145,
    81, 183, 21, 94, 44, 215, 109, 175, 243, 106, 210, 31, 83,
    127, 197, 96, 104, 218, 189, 67, 212, 219, 158, 98, 241, 100,
    37, 38, 195, 200, 244, 242, 76, 47, 4, 214, 78, 20, 162, 164,
    252, 190, 194, 134, 112, 129, 172, 16, 72, 221, 119, 52, 84,
    10, 121, 33, 51, 40, 5, 36, 213, 151, 74, 87, 64, 191, 250,
```

```
    120, 85, 140, 61, 245, 2, 135, 235, 30, 90, 220, 88, 146, 79,
    227, 137, 216, 48, 123, 208, 110, 73, 184, 239, 159, 53, 71,
    17, 27, 93, 136, 251, 77, 144, 223, 11, 182, 57, 107, 185, 75,
    63, 187, 205, 59, 169, 142, 13, 255, 114, 82, 248, 149, 176,
    89, 70, 167, 173, 152, 133, 122, 126, 101, 249, 131, 49, 0, 228,
    19, 111, 45, 161, 41, 128, 124, 196, 203, 117, 206, 141,
    157, 179, 118, 125, 240, 24, 236, 7, 174, 231, 155, 156, 46, 225,
};

UINT Amastrid::_temp[256];

void Amastrid::Initialize(BYTE const* keydata, UINT const& keysize)
{
    UINT kwords[64];
    int keysizew = keysize >> 2;
    if (keysize & 3) keysizew++;
    kwords[keysizew-1] = 0;
    memcpy(kwords, keydata, keysize);
    Bytes2Words(kwords, kwords, keysizew);
    Expansion exp;
    exp.Expand(kwords, keysizew, _kk2);
    exp.GetKK1(_kk1);
    //_kk2 in words
    memcpy(kwords, _kk2, 256);
    Bytes2Words(kwords, kwords, 64);
}

//Generate Initialization Vector
void Amastrid::GenerateIV(UINT* iv)
{
    BYTE temp[4];
    temp[0] = _kk1[(BYTE)_size1]; temp[1] = _kk2[temp[0]];
    temp[2] = _kk1[temp[1]]; temp[3] = _kk2[temp[2]];
    UINT v = Bytes2Word(temp);
    v = KK1(v) ^ KK2(v);
    for (int i = 0; i < (int)_size; i++)
    {
        iv[i] = (v += KK1(KK2(v)));
    }
    Encrypt(iv);
}

 //Encrypt Primitive
void Amastrid::Encrypt(UINT* ar)
{
    UINT c1, c2, c1o, c2o;
    int kpos1, kinc1, kpos2, kinc2;
    int pos1, pos2, pos1o, pos2o, inc1, inc2;
    BYTE temp[4];
    temp[0] = _kk1[(BYTE)_size1]; temp[1] = _kk2[temp[0]];
    temp[2] = _kk1[temp[1]]; temp[3] = _kk2[temp[2]];
    UINT kk1 = KK2(Bytes2Word(temp));
```

```
temp[0] = _kk2[(BYTE)_size1]; temp[1] = _kk1[temp[0]];
temp[2] = _kk2[temp[1]]; temp[3] = _kk1[temp[2]];
UINT kk2 = KK1(Bytes2Word(temp));
int i, k;
for (k = 0; k < (int)_iter; k++)
{
    kpos1 = KK1(kk1);
    kinc1 = Amastrid::F1(KK2(kk2));
    kpos2 = KK2(kk1);
    kinc2 = KK1(Amastrid::F2(kk2));
    kk1 = KK2(kpos1);
    kk2 = KK1(kinc1);
    inc1 = (H1(kinc1) | 1) & _size1;
    inc2 = (H2(kinc2) | 1) & _size1;
    pos1 = kpos1 & _size1;
    if (k > 0)
    {
        if (pos1 != pos2)
        {
            ar[pos1] = Amastrid::F1(ar[pos1]);
            ar[pos2] = Amastrid::G1(ar[pos2]);
            if (pos1 > pos2)
            {
                Amastrid::ShiftRightRot(ar, _size, pos1 - pos2);
            }
            else
            {
                Amastrid::ShiftLeftRot(ar, _size, pos2 - pos1);
            }
        }
    }
    pos2 = kpos2 & _size1;
    if ((inc1 == inc2) && (pos1 == pos2))
    {
        pos2++;
        pos2 &= _size1;
    }
    pos1o = -1;
    pos2o = -1;
    c1o = kk1;
    c2o = kk2;
    for (i = 0; i < (int)_size; i++)
    {
        c1 = ar[pos1];
        c2 = ar[pos2];
        if (pos1 == pos2)
        {
            c1 ^= KK1(c1o);
            c1 += Amastrid::F1(c2o);
            c1 ^= Amastrid::F2(KK2(c2o));
            c1 += Amastrid::G1(c1o);
            c2 = c1;
```

```
            }
            else if (pos1 == pos2o)
            {
                if (pos2 == pos1o)
                {
                    c1 += Amastrid::F2(c2);
                    c1 ^= KK2(c2);
                    c2 ^= Amastrid::F1(c1);
                    c2 += Amastrid::G2(KK1(c1));
                }
                else
                {
                    c1 ^= Amastrid::F1(KK2(c1o));
                    c1 += Amastrid::F2(c2);
                    c2 ^= KK1(c1);
                    c2 += Amastrid::G2(c1o);
                }
            }
            else if (pos2 == pos1o)
            {
                c1 += KK1(c2);
                c1 ^= Amastrid::G2(c2o);
                c2 += Amastrid::F1(KK2(c2o));
                c2 ^= Amastrid::F2(c1);
            }
            else
            {
                c1 ^= KK1(c1o);
                c1 += Amastrid::F2(c2);
                c1 ^= Amastrid::G1(c2o);
                c2 += KK2(c2o);
                c2 ^= Amastrid::F1(c1);
                c2 += c1o;
            }
            ar[pos1] = c1;
            ar[pos2] = c2;
            if (i < (int)_size1)
            {
                c1o = c1;
                c2o = c2;
                pos1o = pos1;
                pos2o = pos2;
                pos1 += inc1;
                pos1 &= _size1;
                pos2 += inc2;
                pos2 &= _size1;
            }
        }
    }
}

 //Decrypt Primitive
```

```
void Amastrid::Decrypt(UINT* ar)
{
    static UINT kk1[10];
    static UINT kk2[10];
    static int kpos1[10];
    static int kinc1[10];
    static int kpos2[10];
    static int kinc2[10];
    UINT c1, c2, c1o, c2o;
    int pos1, pos2, pos1o, pos2o, pos1temp, inc1, inc2;
    BYTE temp[4];
    temp[0] = _kk1[(BYTE)_size1]; temp[1] = _kk2[temp[0]];
    temp[2] = _kk1[temp[1]]; temp[3] = _kk2[temp[2]];
    UINT var1 = KK2(Bytes2Word(temp));
    temp[0] = _kk2[(BYTE)_size1]; temp[1] = _kk1[temp[0]];
    temp[2] = _kk2[temp[1]]; temp[3] = _kk1[temp[2]];
    UINT var2 = KK1(Bytes2Word(temp));
    int i, k;
    for (i = 0; i < (int)_iter; i++)
    {
        kpos1[i] = KK1(var1);
        kinc1[i] = Amastrid::F1(KK2(var2));
        kpos2[i] = KK2(var1);
        kinc2[i] = KK1(Amastrid::F2(var2));
        kk1[i] = (var1 = KK2(kpos1[i]));
        kk2[i] = (var2 = KK1(kinc1[i]));
    }
    for (k = _iter - 1; k >= 0; k--)
    {
        inc1 = _size - ((H1(kinc1[k]) | 1) & _size1);
        inc2 = _size - ((H2(kinc2[k]) | 1) & _size1);
        if (k < (int)_iter - 1)
        {
            pos1temp = pos1;
        }
        pos1 = kpos1[k] & _size1;
        pos2 = kpos2[k] & _size1;
        if ((inc1 == inc2) && (pos1 == pos2))
        {
            pos2++;
            pos2 &= _size1;
        }
        pos1 += inc1;
        pos1 &= _size1;
        pos2 += inc2;
        pos2 &= _size1;
        if (k < (int)_iter - 1)
        {
            if (pos1temp != pos2)
            {
                if (pos1temp > pos2)
                {
```

```
            Amastrid::ShiftLeftRot(ar, _size, pos1temp - pos2);
        }
        else
        {
            Amastrid::ShiftRightRot(ar, _size, pos2 - pos1temp);
        }
        ar[pos1temp] = Amastrid::F1(ar[pos1temp]);
        ar[pos2] = Amastrid::G2(ar[pos2]);
    }
}
for (i = _size1; i >= 0; i--)
{
    if (i > 0)
    {
        pos1o = (pos1 + inc1) & _size1;
        pos2o = (pos2 + inc2) & _size1;
        c1o = ar[pos1o];
        c2o = ar[pos2o];
    }
    else //i == 0
    {
        c1o = kk1[k];
        c2o = kk2[k];
    }
    c1 = ar[pos1];
    c2 = ar[pos2];
    if (pos1 == pos2)
    {
        c1 -= Amastrid::G1(c1o);
        c1 ^= Amastrid::F2(KK2(c2o));
        c1 -= Amastrid::F1(c2o);
        c1 ^= KK1(c1o);
        c2 = c1;
    }
    else if (pos1 == pos2o)
    {
        if (pos2 == pos1o)
        {
            c2 -= Amastrid::G2(KK1(c1));
            c2 ^= Amastrid::F1(c1);
            c1 ^= KK2(c2);
            c1 -= Amastrid::F2(c2);
        }
        else
        {
            c2 -= Amastrid::G2(c1o);
            c2 ^= KK1(c1);
            c1 -= Amastrid::F2(c2);
            c1 ^= Amastrid::F1(KK2(c1o));
            c2o = c1;
        }
    }
```

```
            else if (pos2 == pos1o)
            {
                c2 ^= Amastrid::F2(c1);
                c2 -= Amastrid::F1(KK2(c2o));
                c1 ^= Amastrid::G2(c2o);
                c1 -= KK1(c2);
                c1o = c2;
            }
            else
            {
                c2 -= c1o;
                c2 ^= Amastrid::F1(c1);
                c2 -= KK2(c2o);
                c1 ^= Amastrid::G1(c2o);
                c1 -= Amastrid::F2(c2);
                c1 ^= KK1(c1o);
            }
            ar[pos1] = c1;
            ar[pos2] = c2;
            if (i > 0)
            {
                pos1 = pos1o;
                pos2 = pos2o;
                c1 = c1o;
                c2 = c2o;
            }
        }
    }
}

//Variable block size encryption
void Amastrid::Encrypt(BYTE const* data, BYTE* res, ULONG const& length, int em)
{
    UINT i;
    ULONG lblocks = length / _size4;
    UINT words[BlockSize256];
    UINT words1[BlockSize256];
    if (em == CBC)
    {
        GenerateIV(words1);
    }
    ULONG pos = 0; //current position
    for (ULONG ui=0; ui<lblocks; ui++, pos+=_size4)
    {
        memcpy(words, data+pos, _size4);
        Bytes2Words(words, words, _size);
        if (em == CBC)
        {
            for (i=0; i<_size; i++)
            {
                words[i] ^= words1[i];
            }
```

```
        }
        Encrypt(words);
        if (em == CBC)
        {
            memcpy(words1, words, _size4);
        }
        Words2Bytes(words, res + pos, _size);
    }
    int r = length % _size4;
    if (r > 0)
    {
        memcpy(words, data+pos, r);
        //padding
        BYTE* pb = ((BYTE*)words) + r;
        BYTE v = (BYTE)_size1;
        for (i=0; i<_size4-r; i++, pb++)
        {
            *pb = (v = _kk1[v] + _kk2[(BYTE)i]);
        }
        Bytes2Words(words, words, _size);
        if (em == CBC)
        {
            for (i=0; i<_size; i++)
            {
                words[i] ^= words1[i];
            }
        }
        Encrypt(words);
        Words2Bytes(words, res + pos, _size);
    }
}

//Variable block size decryption
void Amastrid::Decrypt(BYTE const* data, BYTE* res, ULONG const& length, int em)
{
    UINT i;
    ULONG lblocks = length / _size4;
    UINT words[BlockSize256];
    UINT words1[BlockSize256];
    UINT words2[BlockSize256];
    if (em == CBC)
    {
        //IV
        GenerateIV(words1);
    }
    ULONG pos = 0; //current position
    for (ULONG ui=0; ui<lblocks; ui++, pos+=_size4)
    {
        memcpy(words, data+pos, _size4);
        Bytes2Words(words, words, _size);
        if (em == CBC)
        {
```

```
            memcpy(words2, words, _size4);
        }
        Decrypt(words);
        if (em == CBC)
        {
            for (i=0; i<_size; i++)
            {
                words[i] ^= words1[i];
            }
            memcpy(words1, words2, _size4);
        }
        Words2Bytes(words, res + pos, _size);
    }
    int r = length % _size4;
    if (r > 0)
    {
        memcpy(words, data+pos, _size4);
        Bytes2Words(words, words, _size);
        Decrypt(words);
        if (em == CBC)
        {
            for (i=0; i<_size; i++)
            {
                words[i] ^= words1[i];
            }
        }
        //Ignore padding
        Words2Bytes(words, res + pos, r);
    }
}

//Input File Length
ULONG Amastrid::FileLength(ifstream& in)
{
    //Check first the file's state
    if(!in.is_open() || in.bad())
    {
        throw runtime_error("FileLength(), file not opened or in bad state.");
    }
    //Get current position
    streampos currpos = in.tellg();
    //Move to the end
    in.seekg(0, ios::end);
    streampos endpos = in.tellg();
    //Go Back
    in.seekg(currpos, ios::beg);
    return (ULONG)endpos;
}

//Encrypt File
void Amastrid::EncryptFile(string const& infilepath, string const& outfilepath, int em)
{
```

```
if (infilepath.empty() || outfilepath.empty())
{
    throw runtime_error("Amastrid::EncryptFile(), empty file path.");
}
try
{
    UINT i;
    UINT words[BlockSize256];
    UINT words1[BlockSize256];
    ifstream ifs(infilepath.c_str());
    if(!ifs)
    {
        throw runtime_error("Amastrid::EncryptFile(), cannot open input file.");
    }
    ULONG length = FileLength(ifs);
    ULONG lblocks = length / _size4;
    ofstream ofs(outfilepath.c_str());
    if(!ofs)
    {
        throw runtime_error("Amastrid::EncryptFile(), cannot open output file.");
    }
    if (em == CBC)
    {
        //IV
        GenerateIV(words1);
    }
    for (ULONG ui=0; ui<lblocks; ui++)
    {
        ifs.read((char*)words, (streamsize)_size4);
        Bytes2Words(words, words, _size);
        if (em == CBC)
        {
            for (i=0; i<_size; i++)
            {
                words[i] ^= words1[i];
            }
        }
        Encrypt(words);
        if (em == CBC)
        {
            memcpy(words1, words, _size4);
        }
        Words2Bytes(words, (BYTE*)words, _size);
        ofs.write((char*)words, _size4);
    }
    int r = length % _size4;
    if (r > 0)
    {
        ifs.read((char*)words, (streamsize)r);
        //padding
        BYTE* pb = ((BYTE*)words) + r;
        BYTE v = (BYTE)_size1;
```

```
            for (i=0; i<_size4-r; i++, pb++)
            {
                *pb = (v = _kk1[v] + _kk2[(BYTE)i]);
            }
            Bytes2Words(words, words, _size);
            if (em == CBC)
            {
                for (i=0; i<_size; i++)
                {
                    words[i] ^= words1[i];
                }
            }
            Encrypt(words);
            Words2Bytes(words, (BYTE*)words, _size);
            ofs.write((char*)words, _size4);
        }
        ifs.close();
        ofs.close();
    }
    catch (exception const& ex)
    {
        throw runtime_error("Amastrid::EncryptFile(), file read error: "
            + string(ex.what()));
    }
}

//Decrypt File
void Amastrid::DecryptFile(string const& infilepath, string const& outfilepath,
    ULONG const& length, int em)
{
    if (infilepath.empty() || outfilepath.empty())
    {
        throw runtime_error("Amastrid::EncryptFile(), empty file path.");
    }
    try
    {
        UINT i;
        UINT words[BlockSize256];
        UINT words1[BlockSize256];
        UINT words2[BlockSize256];
        ifstream ifs(infilepath.c_str());
        if(!ifs)
        {
            throw runtime_error("Amastrid::EncryptFile(), cannot open input file.");
        }
        ULONG lblocks = length / _size4;
        ofstream ofs(outfilepath.c_str());
        if(!ofs)
        {
            throw runtime_error("Amastrid::EncryptFile(), cannot open output file.");
        }
        if (em == CBC)
```

```
        {
            //IV
            GenerateIV(words1);
        }
        for (ULONG ui=0; ui<lblocks; ui++)
        {
            ifs.read((char*)words, (streamsize)_size4);
            Bytes2Words(words, words, _size);
            if (em == CBC)
            {
                memcpy(words2, words, _size4);
            }
            Decrypt(words);
            if (em == CBC)
            {
                for (i=0; i<_size; i++)
                {
                    words[i] ^= words1[i];
                }
                memcpy(words1, words2, _size4);
            }
            Words2Bytes(words, (BYTE*)words, _size);
            ofs.write((char*)words, _size4);
        }
        int r = length % _size4;
        if (r > 0)
        {
            ifs.read((char*)words, (streamsize)_size4);
            Bytes2Words(words, words, _size);
            Decrypt(words);
            if (em == CBC)
            {
                for (i=0; i<_size; i++)
                {
                    words[i] ^= words1[i];
                }
            }
            Words2Bytes(words, (BYTE*)words, _size);
            //Ignore padding
            ofs.write((char*)words, r);
        }
        ifs.close();
        ofs.close();
    }
    catch (exception const& ex)
    {
        throw runtime_error("Amastrid::EncryptFile(), file read error: " +
            string(ex.what()));
    }
}
```

subsection*4.5 Auxiliary C++ classes for Counter Mode implementation for Version 1.1.0

```
// Counter.h
#ifndef __COUNTER_H__
#define __COUNTER_H__

#include <cstring>

using namespace std;

#define UINT unsigned int

class Counter
{
public:
    //Constructors
    Counter() {}

    Counter(UINT* data, int size)
    {
        Initialize(data, size);
    }

    void Initialize(UINT* data, int size)
    {
        _size = size;
        _size4 = size << 2;
        memcpy(_data, data, _size4);
    }

    UINT& operator[](int i)
    {
        return _data[i];
    }

    UINT* Data()
    {
        return _data;
    }

    //Increment (overload ++)
    void operator++();

private:
    int _size;
    int _size4;
    UINT _data[256];
};

#endif // __COUNTER_H__

// Counter.cpp
#include "Counter.h"
```

```
void Counter::operator++()
{
    static UINT carry;
    static long long res;
    carry = 1;
    register int i = 0;
    while (carry == 1)
    {
        res = (long long)_data[i] + 1;
        _data[i++] = (UINT)res;
        if (i == _size) break;
        carry = (UINT)(res >> 32);
    }
}

// CounterMode.h
#ifndef __COUNTER_MODE_H__
#define __COUNTER_MODE_H__

#include "Counter.h"
#include "Amastrid.h"

class CounterMode
{
public:
    //Constructors
    CounterMode(BYTE const* key, int keysize,
        Amastrid::BlockSize size4=Amastrid::BlockSize16,
        UINT const& iter=3) : _ams(key, keysize, size4, iter),
        _size4(size4), _bcnt(0)
    {
        _size = (_size4 >> 2);
    _wcnt = _size-1;
    _bcnt = 0;
    _ams.GenerateIV(_iv);
    //iv used as a nonce
    _cnt.Initialize(_iv, _size);
    }

    void Reset()
    {
        _wcnt = _size-1;
        _bcnt = 0;
        _cnt.Initialize(_iv, _size);
    }

    void GetNextWord(UINT& rnd);

    void GetNextByte(BYTE& rnd);

    void GetWords(UINT* words, int n)
    {
```

```
        for (register int i = 0; i < n; i++)
        {
            GetNextWord(words[i]);
        }
    }

    void GetBytes(BYTE* bytes, int n)
    {
        for (register int i = 0; i < n; i++)
        {
            GetNextByte(bytes[i]);
        }
    }

private:
    Counter _cnt;
    Amastrid _ams;
    UINT _iv[Amastrid::BlockSize256];
    UINT _bcnt;
    UINT _wcnt;
    int _size;
    int _size4;
};

#endif // __COUNTER_MODE_H__

// CounterMode.cpp
#include "CounterMode.h"

//The counter mode turns a block cipher into a stream cipher. It generates
//the next keystream block by encrypting successive values of a "counter".
//The counter can be any function which produces a sequence which is guaranteed
//not to repeat for a long time, although an actual counter is the simplest and
//most popular. The IV/nonce and the counter can be concatenated, added, or
//XORed together to produce the actual unique counter block for encryption.
void CounterMode::GetNextWord(UINT& rnd)
{
    static UINT words[Amastrid::BlockSize256];
    _wcnt++;
    if (_wcnt == _size)
    {
        _wcnt = 0;
        memcpy(words, _cnt.Data(), _size4);
        _ams.Encrypt(words);
        ++_cnt;
    }
    rnd = words[_wcnt];
}

void CounterMode::GetNextByte(BYTE& rnd)
{
    static UINT word = 0;
```

```
    switch (_bcnt)
    {
        case 0:
            GetNextWord(word);
            rnd = (BYTE)word;
            break;

        case 1:
            rnd = (BYTE)(word >> 8);
            break;

        case 2:
            rnd = (BYTE)(word >> 16);
            break;

        case 3:
            rnd = (BYTE)(word >> 24);
            break;
    }
    _bcnt = (_bcnt + 1) & 3;
}
```