The ATELOPUS Hash Method (Version 2.0.0) - Design and Security Considerations, and Comments to C++ Reference Code

by George Anescu, August 2010
george.anescu@sc-gen.com, www.sc-gen.com

1. Introduction

I was always intrigued why there does not exist any "universal" hashing method with strong cryptographic properties? By a "universal" hashing method I understand a method based on a primitive hash function capable to with an input block of any size and capable to produce a hash result of any size (contraction being assumed obviously). My hashing method ATELOPUS is an atempt to provide such a cryptographic hashing method.

1.1 Design Goals and Characteristics

Simplicity and Mathematical elegance;

Scalability - the capability to work with an input block of any size and to produce a hash result of any size using the same code implementation. The size granularity can be at byte level or at word level.

Dynamic block size - the capability to work with a variable input block size, the input block size being dynamically determined in a pseudo-random manner based on data. By applying a dynamic block size it is eliminated the need for final length padding (also called Merkle - Damgård strengthening), excepting the case of small size messages.

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;

Easy Customization - for example using a keyed ATELOPUS Hash Method inside a company or organization (similar to HMAC methods);

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

Resistence - to clasical and more modern security attacks, including more recent advances like length extension attacks, Joux's multicollisions attacks, fixed-points attacks;

1.2 Limitations

Efficiency - it is definitely a disadvantage compared to other proposals of hashing methods. The first ATELOPUS implementations I tested were working at byte level. They had the advantage of highest possible granularity, but the computing efficiency was low. Staring with Version 2.0.0 ATELOPUS works at word level with the computing speed almost doubled.

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 Main Iterative Construction

ATELOPUS is applying an improved Merkle - Damgård iterative construction. Simply the Merkle - Damgård iterative construction consists in iterating the primitive hash function with a block of message and a chaining value as input (see Fig. 1).

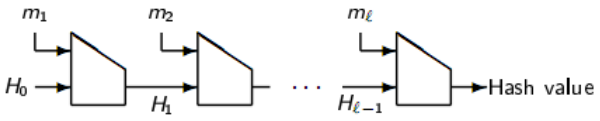

Fig. 1

The modifications consist in the extension of the internal state, the use of a dynamic input block size, and the elimination of length padding to the input message (excepting the case of short messages).

2.2 Extended Internal State

In addtion to the hash value the internal state consist in two word size fast varying variables, defined in the C++ reference code by $\_val1$ and $\_val2$ and a dynamic permutation of the numbers from 0 to 255 maintained in an array of 256 bytes, defined in the C++ reference code by $\_kk1[256]$. The extended internal state is completely hidden, in contrast with the final output hash value which is public. At each iteration, based on the internal state from the previuos message block and on the current message block data, it is generated a new internal state. The permutation is changed dynamically by swapping array positions based on the processed data. This idea was for the first time used in cryptography by the well known stream ciphering method RC4. The initial permutation array is predefined, but the ATELOPUS Hash Method can be easily customized by using a secret initial permutation array.

The idea to extend the internal state of the hash algorithm in order to improve the security is also used in some other existing hash methods, for example the ones applying the Wide-Pipe Hash mode, but the method I present here differs from the Wide-Pipe Hash method.

2.3 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 calculated in a pseudo-random data dependent manner, based on the current values of the variables $\_val1$ and $\_val2$. If we use the notation $min = max/2$, than the block size is calculated as $blocksize = min + ((\_val1 + \_val2) \mod min) + 1$. In the C++ reference code it is implemented in the code block

```
blocksize4 = _val1 + _val2;
blocksize4 &= min1;
blocksize4 |= min;
++blocksize4;
```

in methods $Hash()$ and $HashFile()$, where $min1$ is calculated as $min - 1$. The exceptions to the rule are the case of message length $\leq max$ and the last 2 blocks for messages with length $> max$. If the message

length is $\leq max$ then it is padded to a block of size max. The padding starts with the first byte 0 and then continues with $1, 2, \ldots$, incrementing with 1. If the maximum byte value 255 is atained then the next padding value is reset to 0.

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.

2.4 Iterative Process

For each message block m_i we compute:

$$H_i = h(m_i) \oplus_{kk} H_{i-1}, \;\; i = 1, 2, \ldots l$$

where the function $h(x)$ is the primitive hash function (also called the compression function, because the length of the output hash is usually less then the length of the input block), and the $\oplus_{kk}$ operation is a modified biwise XOR operation for blocks of different sizes. The modifications consist in wrapping around 0 the block of smaller size and mediating the XOR operation through the $\_kk1$ permutation array. In code the $\oplus_{kk}$ operation is implemented by the $XORIPE()$ methods. The wraping around 0 of the smaller block is needed only during the processing of the first message block, therefore most of the time the more efficient overload of the $XORIPE()$ method is used.

2.5 Help Functions

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

```
UINT KK(UINT const& val);
void Swap(UINT const& val1, UINT const& val2);
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);
```

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

The function $Swap()$ is swapping the positions of the $\_kk1$ permutation based on the two word arguments;

$H1()$ 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;

There are also some other help methods in the ATELOPUS class, like the $XORIPE()$ methods already mentioned, and 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.

2.6 Initialization

The initialization algorithm is used in order to generate the initial internal state in a data dependent manner. The initial internal state consists of the H_0 value (which plays the role of an initialization vector - IV), and the initial values of the extension components, the fast varying variables $\_val1$ and $\_val2$ and the permutation array $\_kk1$. The initialization block has the size max, padding being applied for messages less than max. Before the application of the initialization algorithm on the initialization block, the first 8 bytes of the initialization block are bitwise XOR-ed with the message length. It is in order to avoid obvious collisions which can be obtained for messages with length less than max and messages obtainded from them after padding and considered as original messages. The initialization process takes place in a loop applied at least 256 times to the initialization block, but also at least a number of times equal to 2 block lengths ($2 \times max$), whichever is larger. The swapping applied to the $\_kk1$ permutation array is implemented according to the known Durstenfeld shuffle algorithm, which ensure that the $\_kk1$ permutation array is significantly changed by the initialization process. In the C++ referecne code the initialization algorithm is implemented by method $Init()$.

2.7 Primitive Hash Function

The primitive hash function is implemented in the C++ reference code by method $HashPrimitive()$. There are 3 phases in the primitive hash function: 1) propagation of the differences (diffusion and confusion), 2) contraction, and finally 3) combination with the original message block.

2.7.1 Propagation of the Differences

The message is spanned iteratively on its full length a number of times which is determined by the value of $\_iter$ instance variable, which is a parameter initialized in the constructor. By means of the $\_iter$ 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 are involved 4 positions (indexes) in the message block and the values in 2 of them are changed. It is very important for incresed security to make the selection of the positions strongly dependent on the block data (current) and on the already processed message blocks (history). In this way each block is processed on a specific path dependent on its own data and on the processing history imprinted in the extended internal state (the fast varying variables $\_val1$ and $\_val2$, and the permutation array $\_kk1$). Before each data block spanning the initial values of the 4 positions are determined in a pseudo-random manner based on the extended internal state. The increments of two of the indexes have the value 1, while for the other two indexes the they 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 length. 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). Any message block length is prime with any of the prime numbers in array $\_sarrprimes$, because any message block length is less than any of the prime numbers in array $\_sarrprimes$.

Technique: If len is the message block length, and we select in a pseudo-random manner (based on the extended internal state) 2 prime numbers $prime_1$ and $prime_2$ from array $\_sarrprimes$, then it can be easily proved mathematically that the increments $incr_1 = (prime_1 \mod len)$ and $incr_2 = (prime_2 \mod len)$ are also prime with len:

Proof: We can consider by reduction to absurdity that the statement is not true, and that there exists a positive divisor d of both $incr_1$ and len. But we have $prime_1 = q_1 len + incr_1$ for some positive integer q_1, and from it we get that d should also divide $prime_1$, which is a contradiction because $prime_1$ is a prime number by hypothesis, q.e.d.

By calculating the increments $incr_1$ and $incr_2$ using the presented technique we can be sure that we obtain the full coverage property.

After the first full length spanning of the message block we introduce a data difference based on the hash output size given by variable $\_size$. We apply the XOR operation between the first word in the array and the $\_size$ mediated by the permutation array $\_kk1$, as can be seen in the code block:

```
if (k == 1)
{
    //Introduce a data difference based on _size
    ar[0] ^= KK(_size);
}
```

The purpose of this data difference is to make sure that we obtain different results after the first phase (propagation of the differences) when we process the same data block with different hash output sizes.

2.7.2 Contraction

The purpose of the contraction phase is to process the data block resulted from the previous phase (propagation of the differences) in order to obtain a result block having the size equal with the hash output size (determined by the instance variable $\_size$). This process is usually a contraction because usually $\_size$ is less than len, but the algorithm, as it is implemented, can also work for expansions. The number of iterations in the contraction loop is al least $len/2 + 1$, in this way ensuring that all the block positions are involved in the process. The same technique for calculating the indexes of the processed data in a pseudo-random manner, as the one applied in the first phase, is also applied here for both the input data block and the output data block.

2.7.3 Combination with the Original

Here the data block resulted from the previous phase (contraction) is combined with the original message block in a manner that involves the extended internal state. In this phase no further swapping is applied on $\_kk1$ permutation array. The same technique for calculating the indexes of the processed data in a pseudo-random manner, as the one applied in the first two phases, is also applied here for both the resut data block and the original data block.

3. Security Considerations

3.1 First and Second Image Attacks

There are two types of preimage attacks: First Preimage Attack, when given a hash result H it is required to find the original message m such that $hash(m) = H$, where $hash$ is the hash function; and Second Preimage Attack with the weaker requirement that given a fixed message m_1 to find a different message m_2 such that $hash(m_2) = hash(m_1)$.

The most important property of a primitive hash function is to be difficult to reverse. For my ATELOPUS method the difficulty to reverse is increased by the use of the extended internal state, which is unknown to the attacker, the only data the attacker knows being the hash result H. The search space of the extended internal state is very large. Only for the $\_kk1$ permutation array the dimension of the search space is 256!, which can be approximated to 10×2^{1680} by using the well known Stirling's formula $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$. Additionally, considering the serach space of the fast varying variables $\_val1$ and $\_val2$, which is 2^{64}, we get an enormous total search space of dimension $\approx 10 \times 2^{1744}$, which is far beyond the capabilities of the present day computers.

Additionally to the above considerations, as it was shown in section 2.7, during the third phase of the ATELOPUS primitive hash function, called combination with the original, the original message is combined with the hash result obtained so far in a manner mediated by the extended internal state of the algorithm. This technique adds additional difficulty to reversing the ATELOPUS primitive hash function, because in order to start the reversing process you also need to guess the original message.

3.2 Collisions

The requirements of a Collision Attack are weaker than the ones of the Second Preimage Attack, a Collision Attack involving finding two arbitrarily different messages m_1 and m_2 such that $hash(m_2) = hash(m_1)$.

I am not aware of any theoretical weakness of the ATELOPUS algorithm which can provide a method to generate collisions more efficiently than the known Birthday Attack, which has a complexity of about $2^{\text{-}size/2}$, where $\_size$ is the size of the hash output in bits. It means that for a hash output of 32 bytes ($32 \times 8 = 256$ bits), by applying the birthday attack we need to calculate about 2^{128} message samples in order to get a collision with probability 0.5. This is beyond the power and stoarge capabilities of the present day computers and is considered acceptable by the cryptography community.

I did only some stochastic tests, by flipping randomly some bits of a message, calculating the hash and comparing with the hash of the original message. I was not able to find any collision in this way.

3.3 Linear and Differential Cryptanalysis

I expect that ATELOPUS is resistent to Linear and Differential Cryptanalysis attacks due to the nonlinearity and dynamic caracteristics of the $\_kk1$ permutation array and due to tha fact that the processing path is data dependent.

3.4 Joux's Multicollisions Attack

The Joux's Multicollisions Attack shows how you can compute a 2^k multicollision for k times the cost of finding k simple collisions. As a simple example, assume that we were able to find 2 simple collisions:

$$hash(H_0, M_1) = hash(H_0, M_1^{'}) = H_1$$

$$hash(H_1, M_2) = hash(H_1, M_2^{'}) = H_2$$

where H_0 is the initialization vector for the first collision, and H_1 the result of the first collision is the initialization vector for the second collision. It is easy to see that we can construct a multicollision of order 4:

$$hash(H_0, M_1 \| M_2) = hash(H_0, M_1^{'} \| M_2) = hash(H_0, M_1 \| M_2^{'}) = hash(H_0, M_1^{'} \| M_2^{'})$$

where the $\|$ means string concatenation.

In case of ATELOPUS such an attack is difficult to conduct. First, the initial internal state does not consist only in the initialization vector, it includes additionally the extended internal state's components and both the initialization vector and the extended internal state are dependent on a first portion of the message, so that they cannot be arbitrarily set. Second, for ATELOPUS this kind of attack works only if we are able to find simple collisions with the same hash result and additionally with the same extended internal state Ext, for the simple example we need:

$$hash(H_0, Ext_0, M_1) = hash(H_0, Ext_0, M_1^{'}) = (H_1, Ext_1)$$

$$hash(H_1, Ext_1, M_2) = hash(H_1, Ext_1, M_2^{'}) = (H_2, Ext_2)$$

Such collisions are very difficult to obtain considering the large dimension of the extended internal state's space.

3.5 Fixed Point Attack

We have a fixed point when the initialization vector is equal to the hash result:

$$hash(H, M) = H$$

If we are able to find a fixed point than we can easily produce multicollisions accordin to:

$$hash(H, M) = hash(H, M\|M) = hash(H, M\|M\|M) = \ldots = hash(H, M\|M\|\ldots\|M) = \ldots = H$$

In case of ATELOPUS it is difficult to find such a fixed point because it would also require the same extended internal state Ext:

$$hash(H, Ext, M) = (H, Ext)$$

and, as was shown in 3.1, the dimension of the extended internal state's space is very large.

3.5 Length Extension Attack

This type of attack is usually applied to HMACs which are Hash-based Message Authentication Codes, i.e. hashes with secret keys. Usually a HMAC is obtained from a hash method by prefixing the message with a key:

$$hash(H_0, K\|M) = H_1$$

The idea of this attack is that the attacker can calculate HMACs without knowing the secret key K by simply knowing a HMAC result and using it as an initialization vector. If x is a text the attacker chooses, then we have:

$$hash(H_0, K\|M\|x) = hash(H_1, x)$$

or to be more rigurous, some padding may be required for hash methods using Merkle length padding (it is not the case for ATELOPUS):

$$hash(H_0, K\|M\|p\|x) = hash(H_1, x)$$

with p being the length padding block.

In case of ATELOPUS such an attack cannot be applied for two reasons, first ATELOPUS being a dynamic block size method there is no guarantee that after you append a text to a message the sizes of the final blocks are computed identically as for the original message, and second you also need the extended internal state of the known HMAC in order to calculate $hash(H_1, x)$, but the extended internal state is not public.

4.0 Apendix

4.1 Test Samples

hashsize=32, iter=3, max blocksize=128:

1)
data=""
hexresult="51CA7440E9F7E6A1E77B32CD653C1131571D690D939ADF3185D1977DC91369B2"

2)
data="AABB"
hexresult="09A007AB91986399D047E55399D76C542A700712D3947FA2A21ED0CE8B28BE3E"

3)
data="AAAABBBBAAAABBBB"
hexresult="E940F40E8502154A20047DB7F924B6FE87ACC46ABDC65673C38C8B99AD4062A5"

4)
data="AAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBB
BAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBB"
hexresult="B60F0B3F3CD0033B07A1FEF9D22269CC6C277928A03B17C6C130055BE941C4BA"

5)
data="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYY"
hexresult="80ED1BE8212BE95921A32A659DB990E12C5FE593546038A9EDCA5D526806172D"

6)
data="XYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYY"
hexresult="C4C05D1900083254358672292175E7945174E5100F29811E71706BC514068588"

7)
data="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYY"
hexresult="517F51D71165CEEF657DC90408F595F58C030AC9071EFE5A372FE18E1BD2982E"

hashsize=64, iter=3, max blocksize=256

1)
data=""
hexresult="AC157DEAB310A3A6DF0D523877C4600FFF0F007E51EC30897E3AA9858F291569F3A6
7CAFCD9C563D1C1374C045867E0064AF01EDAA72314001F1E1865208BA8B"

2)
data="AABB"
hexresult="70329BCD6498ECB1EA1FC67DA0858E57330CF693E93AD1C6B94F026D9457C9965C9D
2E9EE6E138D35993CEABA4345EAA33338D0D37AEFC89DD948C38FA597B05"

3)
data="AAAABBBBAAAABBBB"
hexresult="42BD9F90B17405CD764993EED3AF2C1BFDECFD5DAAC64A85A2CC65D71B5A075E6BCF
0151CEA2193CCFBD81FF0BDF8E7F8976A1D47E15E9D0A15732F4EDAECFB6"

4)
data="AAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBB
BAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBBAAAABBBB"
hexresult="36A8997FFC2AACBA9CAE792B0D4A480B4E9AC0F1DBA568EF433C95697D04A6B0A6CC
475F566E30EA6D7541CEF1FF1C5880AD84AFF8EB851A685AB73D6442767D"

5)
data="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYY"
hexresult="835BF9E8F46EDC871B4B549F76091E2D6420545D34089807F28A7EE717E90E5DA0A4
4FB8168EF2A53B77B47969798DFC6C3290619F883AA58056D46DBCFD83D2"

6)
data="XYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYY"
hexresult="F24E79788C3FFE72159F60E969BC81DE08A33BFFF2EE21807A76AAED44434DE1CFDE
7FCE2C4BDF87BC9FC18F4D8C04773054039B8C52D259578330AAD2B03206"

7)
data="YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY"
hexresult="3A206A178B8E206BDF927B335D2BDF54A5E5845276AA13C46A67FAF9A5F8AD5E79DA

D7F2CC27A464CDC072410D4D717EBB25428718B1528EE7A890461C91E4ED”

4.2 C++ Reference Code

```
// Atelopus.h
#ifndef __ATELOPUS_H__
#define __ATELOPUS_H__

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

using namespace std;

#define BYTE unsigned char
#define UINT unsigned int

class Atelopus
{
//**************************************************************************
//
// The ATELOPUS hashing method, Version 2.0.0 (28 August 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
// This is the C++ implementation of a new cryptographic hashing method called ATELOPUS,
// which can be considered an universal hashing method. It can work with dynamic sizes of
// the input data blocks (at word granularity) and is capable to generate hash outputs
// covering a practical domain of sizes (at word granularity).
//
// COPYRIGHT PROTECTION: The ATELOPUS hashing 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:
    //Block Sizes in bytes
    //BlockSize16   - variable between 9 and 16
    //BlockSize32   - variable between 17 and 32
    //BlockSize64   - variable between 33 and 64
    //BlockSize128  - variable between 65 and 128
    //BlockSize256  - variable between 129 and 256
    //BlockSize512  - variable between 257 and 512
    //BlockSize1024 - variable between 513 and 1024
```

```
enum AtelopusBlockSize
{
    BlockSize16 = 16, BlockSize32 = 32, BlockSize64 = 64, BlockSize128 = 128,
    BlockSize256 = 256, BlockSize512 = 512, BlockSize1024 = 1024
};

//Constructors
Atelopus(int size4, UINT iter, AtelopusBlockSize bs4=BlockSize16) : _iter(iter),
    _size4(size4), _bs4(bs4)
{
    _bs = _bs4 >> 2; //block size in words
    _size = _size4 >> 2; //block size in bytes
    if (Atelopus::IsBigEndian())
    {
        Swap = &Atelopus::SwapBE;
        Bytes2Word = Atelopus::Bytes2WordBE;
        Word2Bytes = Atelopus::Word2BytesBE;
    }
    else
    {
        Swap = &Atelopus::SwapLE;
        Bytes2Word = Atelopus::Bytes2WordLE;
        Word2Bytes = Atelopus::Word2BytesLE;
    }
    Reset();
}

void Reset()
{
    memcpy(_kk1, _skk1, 256);
    BYTE temp[4];
    temp[0] = _kk1[0];
    temp[1] = _kk1[64];
    temp[2] = _kk1[128];
    temp[3] = _kk1[192];
    _val1 = Bytes2Word(temp);
    temp[0] = _kk1[32];
    temp[1] = _kk1[96];
    temp[2] = _kk1[160];
    temp[3] = _kk1[224];
    _val2 = Bytes2Word(temp);
}

//Hash Primitive
void HashPrimitive(UINT const* data, UINT* res, int const& len);

//Dynamic block size hashing
void Hash(BYTE const* data, BYTE* res, unsigned long long const& length);

//Dynamic block size hashing for a file
void HashFile(string const& filepath, BYTE* res);
```

```
private:
    UINT KK(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;
        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;
    }

    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);
}

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));
}

//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);
}

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

void Init(UINT* ar, int const& len);

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);
    }
}

//In place XOR extended
void XORIPE(UINT* ar1, UINT const* ar2, int const& len)
{
    for (register int i=0; i<len; i++)
    {
        ar1[i] ^= KK(ar2[i]);
    }
```

```
    }

    //In place XOR extended
    void XORIPE(UINT* ar1, UINT const* ar2, int const& len1, int const& len2)
    {
        int max = (len1 > len2) ? len1 : len2;
        for (register int i=0,j=0,k=0; i<max; i++,j++,k++)
        {
            if (j == len1) j = 0;
            if (k == len2) k = 0;
            ar1[j] ^= KK(ar2[k]);
        }
    }

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

    const static BYTE _skk1[256];
    const static short _sarrprimes[256];
    UINT _iter; //hash primitive rounds
    UINT _size; //digest size in words
    UINT _size4; //digest size in bytes
    UINT _bs; //block size in words
    UINT _bs4; //block size in bytes
    UINT _val1, _val2;
    BYTE _kk1[256];
};

#endif // __ATELOPUS_H__

// Atelopus.cpp
#include "Atelopus.h"

//**************************************************************************
//
// The ATELOPUS hashing method, Version 2.0.0 (28 August 2010)
// Copyright (C) 2009-2010, George Anescu, www.sc-gen.com
// All right reserved.
//
//**************************************************************************

const BYTE Atelopus::_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,
};

const short Atelopus::_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,
};

void Atelopus::HashPrimitive(UINT const* data, UINT* res, int const& len)
{
    //Copy in work buffer
    UINT ar[BlockSize256];
    memcpy(ar, data, len<<2);
    int len1 = len - 1;
    int lend2 = len >> 1;
    //1) Propagate differences
    register UINT temp1, temp2;
```

```
register int i, k, ix, ix1, pos1, pos2, incr1, incr2;
for (k = 0; k < (int)_iter; k++)
{
    if (k == 1)
    {
        //Introduce a data difference based on _size
        ar[0] ^= KK(_size);
    }
    ix = H1(Atelopus::F1(_val1) ^ KK(_val2)) % len;
    ix1 = ix + lend2;
    if (ix1 >= len) ix1 -= len;
    incr1 = _sarrprimes[H1(KK(_val1) ^ _val2)];
    incr2 = _sarrprimes[H2(_val1 + KK(_val2))];
    incr1 %= len;
    incr2 %= len;
    pos1 = H1(Atelopus::G2(_val1)) % len;
    pos2 = H2(Atelopus::F2(_val2)) % len;
    if ((incr1 == incr2) && (pos1 == pos2))
    {
        pos2++;
        if (pos2 == len) pos2 = 0;
    }
    for (i=0; i<len; i++)
    {
        if (pos1 == pos2)
        {
            temp1 = ar[pos1];
            _val1 ^= KK(temp1);
            _val1 += ar[ix];
            _val2 ^= Atelopus::F1(_val1);
            _val2 += temp1;
            _val2 ^= ar[ix1];
            (this->*Swap)((temp1 = Atelopus::G1(temp1) ^ _val1), KK(_val2));
            ar[pos1] = Atelopus::F2(KK(temp1)) + _val2;
        }
        else
        {
            temp1 = ar[pos1];
            temp2 = ar[pos2];
            _val1 ^= KK(temp1);
            _val1 += ar[ix];
            _val2 ^= Atelopus::F1(_val1);
            _val2 += temp2;
            _val2 ^= ar[ix1];
            (this->*Swap)((ar[pos1] = Atelopus::G1(temp1) ^ _val1), KK(_val2));
            ar[pos2] = Atelopus::F2(KK(temp2)) + _val2;
        }
        if (i < len1)
        {
            ix++;
            if (ix == len) ix = 0;
            ix1++;
```

```
            if (ix1 == len) ix1 = 0;
            pos1 += incr1;
            if (pos1 >= len) pos1 -= len;
            pos2 += incr2;
            if (pos2 >= len) pos2 -= len;
        }
    }
}
//2) Contracting (assuming _size < len, but it can also expand)
incr1 = _sarrprimes[H1(KK(_val1) ^ _val2)];
incr2 = _sarrprimes[H2(_val1 + KK(_val2))];
incr1 %= len;
incr2 %= len;
pos1 = H1(Atelopus::G2(_val1)) % len;
pos2 = H2(Atelopus::F2(_val2)) % len;
if ((incr1 == incr2) && (pos1 == pos2))
{
    pos2++;
    if (pos2 == len) pos2 = 0;
}
int ik, pk;
ik = _sarrprimes[H1(KK(Atelopus::F1(_val1)) + _val2)] % _size;
pk = H1(Atelopus::F1(_val1) ^ KK(_val2)) % _size;
int max = (lend2 < (int)_size) ? _size : (lend2+1);
register bool over = false;
int size1 = _size - len;
for (i=0,k=pk,ix=lend2,ix1=0; ix1<max; i++,k+=ik,ix++,ix1++,pos1+=incr1,pos2+=incr2)
{
    if (i == len)
    {
        i = 0;
        size1 -= len;
    }
    if (k >= (int)_size)
    {
        k -= _size;
        if (k == pk) over = true;
    }
    if (ix == len) ix = 0;
    if (pos1 >= len) pos1 -= len;
    if (pos2 >= len) pos2 -= len;
    temp1 = ar[pos1];
    _val1 ^= KK(temp1);
    _val1 += ar[i];
    _val2 ^= Atelopus::F1(_val1);
    if (over)
    {
        res[k] += _val2;
    }
    else
    {
        res[k] = _val2;
```

```
        }
        _val2 += ar[pos2];
        _val2 ^= ar[ix];
        if (size1 > 0)
        {
            //only for expansion
            ar[i] ^= KK(_val2);
            ar[ix] += Atelopus::F2(_val1);
        }
        (this->*Swap)(Atelopus::G2(temp1) ^ KK(_val1), _val2);
    }
    //3) Combining with the original
    incr1 = _sarrprimes[H1(KK(_val1))] % len;
    incr2 = _sarrprimes[H2(KK(_val2))] % _size;
    i = H1(Atelopus::F1(_val1)) % len;
    k = H2(Atelopus::G2(_val2)) % _size;
    max = (len < (int)_size) ? _size : len;
    for (ix = 0; ix < max; ix++)
    {
        temp1 = data[i];
        _val1 += Atelopus::F1(KK(temp1));
        _val2 ^= _val1 + temp1;
        res[k] ^= _val2;
        i += incr1;
        if (i >= len) i -= len;
        k += incr2;
        if (k >= (int)_size) k -= _size;
    }
}

void Atelopus::Init(UINT* ar, int const& len)
{
    register int pos1 = 0;
    register int pos2 = (len >> 2);
    register int pos3 = (len >> 1);
    register int pos4 = pos2 + pos3;
    if ((len & 0x3) == 3) //4k+3
    {
        pos2++;
        pos3++;
        pos4++;
    }
    register UINT val1 = _val1 ^ len;
    register UINT val2 = _val2 + Atelopus::F1((BYTE)len);
    BYTE temp[4];
    temp[0] = _kk1[16]; temp[1] = _kk1[80]; temp[2] = _kk1[144]; temp[3] = _kk1[208];
    register UINT val3;
    val3 = Bytes2Word(temp);
    val3 ^= Atelopus::F2((BYTE)len);
    temp[0] = _kk1[48]; temp[1] = _kk1[112]; temp[2] = _kk1[176]; temp[3] = _kk1[240];
    register UINT val4;
    val4 = Bytes2Word(temp);
```

```
    val4 += (BYTE)len;
    UINT *pui1, *pui2, *pui3, *pui4, temp1;
    //at least 2 rounds
    int len2 = len<<1;
    int max = (len2 > 256) ? len2 : 256;
    for (register int i = 0; i < max; i += 4)
    {
        pui1 = &ar[pos1]; *pui1 ^= val1; *pui1 += KK(val2);
        pui2 = &ar[pos2]; *pui2 ^= val2; *pui2 += KK(val3);
        pui3 = &ar[pos3]; *pui3 ^= val3; *pui3 += KK(val4);
        pui4 = &ar[pos4]; *pui4 ^= val4; *pui4 += KK(val1);
        val1 ^= *pui1; val1 += Atelopus::F1(*pui2); val1 ^= KK(*pui3);
        val2 ^= *pui2; val2 += Atelopus::F2(*pui3); val2 ^= KK(*pui4);
        val3 ^= *pui3; val3 += Atelopus::F1(*pui4); val3 ^= KK(*pui1);
        val4 ^= *pui4; val4 += Atelopus::F2(*pui1); val4 ^= KK(*pui2);
        temp[0] = (BYTE)i; temp[1] = (BYTE)(i+1); temp[2] = (BYTE)(i+2); temp[3] = (BYTE)(i+3);
        temp1 = Bytes2Word(temp);
        (this->*Swap)(temp1, Atelopus::G1((val1 + val2)^(val3 + val4)));
        pos1++;
        if (pos1 == len) pos1 = 0;
        pos2++;
        if (pos2 == len) pos2 = 0;
        pos3++;
        if (pos3 == len) pos3 = 0;
        pos4++;
        if (pos4 == len) pos4 = 0;
    }
    _val1 = val1 ^ val2; _val1 += val3;
    _val2 = val2 + val3; _val2 ^= val4;
}

//Dynamic block size hashing (it is not using padding, excepting messages with length less
//than the maximum block size)
void Atelopus::Hash(BYTE const* data, BYTE* res, unsigned long long const& length)
{
    //calculate min, max, double max
    int max = _bs4;
    int maxw = max >> 2;
    int min = max >> 1;
    int max2 = max << 1;
    //work buffers
    UINT ar[BlockSize256];
    UINT ar1[BlockSize256];
    register int i, blocksize, blocksize4;
    BYTE* pbytes;
    if (length <= (unsigned long long)max)
    {
        //Just one block
        blocksize4 = max;
        blocksize = max >> 2;
        //Padding
        pbytes = (BYTE*)ar + (int)length;
```

```
    for (i = 0; i < blocksize4-(int)length; i++,pbytes++)
    {
        *pbytes = (BYTE)i;
    }
    memcpy(ar, data, (size_t)length);
    memcpy(ar1, ar, blocksize4);
    //XOR with the length (no more than 2 bytes)
    int temp = (int)length;
    pbytes = (BYTE*)ar1;
    *pbytes ^= (BYTE)temp;
    temp >>= 8;
    *(++pbytes) ^= (BYTE)temp;
    Bytes2Words(ar, ar, blocksize);
    Bytes2Words(ar1, ar1, blocksize);
    Init(ar1, blocksize);
    HashPrimitive(ar, (UINT*)res, blocksize);
    XORIPE((UINT*)res, ar1, _size, blocksize);
    //Result in bytes
    Words2Bytes((UINT*)res, res, _size);
    return;
}
int min1 = min - 1;
unsigned long long pos = 0;
unsigned long long len1;
register bool init = false;
while (true)
{
    len1 = length - pos; //remaining bytes
    if (len1 <= (unsigned long long)max2)
    {
        //Second to last block
        blocksize4 = (int)len1;
        blocksize4 >>= 1;
        blocksize = blocksize4 >> 2;
        blocksize4 = blocksize << 2;
        memcpy(ar, data+pos, (size_t)blocksize4);
        if (!init)
        {
            memcpy(ar1, data, (size_t)max);
            //XOR with the length (no more than 2 bytes)
            unsigned long long temp = length;
            pbytes = (BYTE*)ar1;
            *pbytes ^= (BYTE)temp;
            temp >>= 8;
            *(++pbytes) ^= (BYTE)temp;
            Bytes2Words(ar1, ar1, maxw);
            Init(ar1, maxw);
            Bytes2Words(ar, ar, blocksize);
            HashPrimitive(ar, (UINT*)res, blocksize);
            XORIPE((UINT*)res, ar1, _size, maxw);
        }
        else
```

```
    {
        HashPrimitive(ar, (UINT*)res, blocksize);
        XORIPE((UINT*)res, ar1, _size);
    }
    memcpy(ar1, res, (size_t)_size4);
    //Last block
    pos += blocksize4;
    blocksize4 = (int)(len1 - blocksize4);
    blocksize = blocksize4 >> 2;
    if ((blocksize4 & 3) != 0)
    {
        ar[blocksize] = 0;
        blocksize++;
    }
    memcpy(ar, data+pos, (size_t)blocksize4);
    Bytes2Words(ar, ar, blocksize);
    HashPrimitive(ar, (UINT*)res, blocksize);
    XORIPE((UINT*)res, ar1, _size);
    //Result in bytes
    Words2Bytes((UINT*)res, res, _size);
    return;
}
else
{
    if (!init)
    {
        memcpy(ar1, data, (size_t)max);
        //XOR with the length
        unsigned long long temp = length;
        pbytes = (BYTE*)ar1;
        *pbytes ^= (BYTE)temp;
        for (i = 1; i < 8; i++)
        {
            temp >>= 8;
            *(++pbytes) ^= (BYTE)temp;
        }
        Bytes2Words(ar1, ar1, maxw);
        Init(ar1, maxw);
    }
    blocksize4 = _val1 + _val2;
    blocksize4 &= min1;
    blocksize4 |= min;
    ++blocksize4;
    blocksize = blocksize4 >> 2;
    blocksize4 = blocksize << 2;
    memcpy(ar, data+pos, (size_t)blocksize4);
    Bytes2Words(ar, ar, blocksize);
    HashPrimitive(ar, (UINT*)res, blocksize);
    if (!init)
    {
        XORIPE((UINT*)res, ar1, _size, maxw);
        init = true;
```

```
            }
            else
            {
                XORIPE((UINT*)res, ar1, _size);
            }
            memcpy(ar1, res, (size_t)_size4);
            pos += blocksize4;
        }
    }
}

//Input File Length
unsigned long long Atelopus::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 (unsigned long long)endpos;
}

//Dynamic block size hashing for a file
void Atelopus::HashFile(string const& filepath, BYTE* res)
{
    if (filepath.empty())
    {
        throw runtime_error("Atelopus::HashFile(), empty file path.");
    }
    try
    {
        //calculate min, max, double max
        int max = _bs4;
        int maxw = max >> 2;
        int min = max >> 1;
        int max2 = max << 1;
        ifstream fs(filepath.c_str());
        if(!fs)
        {
            throw runtime_error("Atelopus::HashFile(), cannot open file.");
        }
        unsigned long long length = FileLength(fs);
        //work buffers
        UINT ar[BlockSize256];
        UINT ar1[BlockSize256];
```

```
register int i, blocksize, blocksize4;
BYTE* pbytes;
if (length <= (unsigned long long)max)
{
    //Just one block
    blocksize4 = max;
    blocksize = max >> 2;
    //Padding
    pbytes = (BYTE*)ar + (int)length;
    for (i = 0; i < blocksize4-(int)length; i++,pbytes++)
    {
        *pbytes = (BYTE)i;
    }
    if (length > 0)
    {
        fs.read((char*)ar, (streamsize)length);
    }
    memcpy(ar1, ar, blocksize4);
    //XOR with the length (no more than 2 bytes)
    int temp = (int)length;
    pbytes = (BYTE*)ar1;
    *pbytes ^= (BYTE)temp;
    temp >>= 8;
    *(++pbytes) ^= (BYTE)temp;
    Bytes2Words(ar, ar, blocksize);
    Bytes2Words(ar1, ar1, blocksize);
    Init(ar1, blocksize);
    HashPrimitive(ar, (UINT*)res, blocksize);
    XORIPE((UINT*)res, ar1, _size, blocksize);
    //Result in bytes
    Words2Bytes((UINT*)res, res, _size);
    fs.close();
    return;
}
int min1 = min - 1;
unsigned long long pos = 0;
unsigned long long len1;
register bool init = false;
while (true)
{
    len1 = length - pos; //remaining bytes
    if (len1 <= (unsigned long long)max2)
    {
        //Second to last block
        blocksize4 = (int)len1;
        blocksize4 >>= 1;
        blocksize = blocksize4 >> 2;
        blocksize4 = blocksize << 2;
        if (!init)
        {
            fs.read((char*)ar1, (streamsize)max);
            //Reset file position
```

```
            fs.seekg(0, ios::beg);
            //XOR with the length (no more than 2 bytes)
            unsigned long long temp = length;
            pbytes = (BYTE*)ar1;
            *pbytes ^= (BYTE)temp;
            temp >>= 8;
            *(++pbytes) ^= (BYTE)temp;
            Bytes2Words(ar1, ar1, maxw);
            Init(ar1, maxw);
            fs.read((char*)ar, (streamsize)blocksize4);
            Bytes2Words(ar, ar, blocksize);
            HashPrimitive(ar, (UINT*)res, blocksize);
            XORIPE((UINT*)res, ar1, _size, maxw);
        }
        else
        {
            fs.read((char*)ar, (streamsize)blocksize4);
            HashPrimitive(ar, (UINT*)res, blocksize);
            XORIPE((UINT*)res, ar1, _size);
        }
        memcpy(ar1, res, (size_t)_size4);
        //Last block
        pos += blocksize4;
        blocksize4 = (int)(len1 - blocksize4);
        blocksize = blocksize4 >> 2;
        if ((blocksize4 & 3) != 0)
        {
            ar[blocksize] = 0;
            blocksize++;
        }
        fs.read((char*)ar, (streamsize)blocksize4);
        Bytes2Words(ar, ar, blocksize);
        HashPrimitive(ar, (UINT*)res, blocksize);
        XORIPE((UINT*)res, ar1, _size);
        //Result in bytes
        Words2Bytes((UINT*)res, res, _size);
        fs.close();
        return;
    }
    else
    {
        if (!init)
        {
            fs.read((char*)ar1, (streamsize)max);
            //Reset file position
            fs.seekg(0, ios::beg);
            //XOR with the length
            unsigned long long temp = length;
            pbytes = (BYTE*)ar1;
            *pbytes ^= (BYTE)temp;
            for (i = 1; i < 8; i++)
            {
```

```
                    temp >>= 8;
                    *(++pbytes) ^= (BYTE)temp;
                }
                Bytes2Words(ar1, ar1, maxw);
                Init(ar1, maxw);
            }
            blocksize4 = _val1 + _val2;
            blocksize4 &= min1;
            blocksize4 |= min;
            ++blocksize4;
            blocksize = blocksize4 >> 2;
            blocksize4 = blocksize << 2;
            fs.read((char*)ar, (streamsize)blocksize4);
            Bytes2Words(ar, ar, blocksize);
            HashPrimitive(ar, (UINT*)res, blocksize);
            if (!init)
            {
                XORIPE((UINT*)res, ar1, _size, maxw);
                init = true;
            }
            else
            {
                XORIPE((UINT*)res, ar1, _size);
            }
            memcpy(ar1, res, (size_t)_size4);
            pos += blocksize4;
        }
    }
}
catch (exception const& ex)
{
    throw runtime_error("Atelopus::HashFile(), file read error: " + string(ex.what()));
}
}
```