The Mumblepad Block Cipher
version 1.0, March 14, 2017
Created by Kyle Granger

1 Introduction

Mumblepad is a symmetric-key, block cipher algorithm. It is implemented as a C++ software library, easily linkable to applications.

The main features of Mumblepad are:

- a relatively large 4096-byte key.
- padding of plaintext blocks with random number bytes, enabling completely different encrypted blocks containing same plaintext.
- no requirement of a block cipher mode, thus parallelizable.
- implementation on GPU, in addition to CPU.
- may work with 6 different block sizes, ranging from 128 to 4096 bytes.
- the multi-threaded implementation encrypts or decrypts @ 210MB/s on laptop.

On a GPU, encryption and decryption are performed by a pixel shader, or more accurately, an OpenGL fragment shader. Each round of diffusion and confusion is executed essentially by drawing two triangles. This may be on a PC, smartphone or tablet.

1.1 Key size

Large 4096-byte, 32768-bit key. The size of key is constant, regardless of the encryption block size.

1.2 Padded Plaintext

The plaintext is mixed with a small amount of random bytes, generatated by a pseudorandom number generator (RC4). Additionally, there are 8 bytes of metadata in each block: 32-bit checksum, a 16-bit length field, and a 16-bit sequence number. The percentage of non-plaintext data ranges from 2.4% to 12.5% of a block, depending on block size.

The advantage here, is that encrypted blocks containing the same plaintext are completely different from one another, as they contain different pseudorandom bytes as padding. The resulting encrypted blocks are as mathematically different from each other as two same-sized blocks of random numbers. I.e., half of the corresponding bits are different, and 255/256 of the bytes are different. This works even with 2.4% padding: 4000 bytes plaintext in 4096-byte block.

1.3 Six block sizes

Mumblepad may be configured to work with six different block sizes. These different configurations should be considered as different block ciphers: they do not interoperate with each other, even thought they may use the same key. Other than that, they differ only with respect to block size. In addition to a block size of 4KB, there are 2KB, 1KB, 512-byte, 256-byte, and 128-

byte. The varying block sizes correspond to the varying number graphics texture rows the cipher uses in the GPU implementation. Each row is 32 pixels, which is 128 bytes or 1024 bits. One can have 1, 2, 4, 8, 16, or 32 rows.

There is also no on-the-fly adaptive behaviour which can change block size on a per-block basis. One simply chooses the variant which fits best for the given application.

1.4 Rounds and passes

During encryption, 8 rounds are performed, each with their own set of tables and textures. First a diffusion step is done, followed by a confusion step. During decryption, the process is reversed.

1.5 Diffusion

Diffusion is the process of moving bits from one position to another. This is perhaps the most complicated bit of the Mumblepad block cipher algorithm. More explicitly here than elsewhere in the code, do we look at working data in a particular stage as texture data, i.e., pixels. 32x32 pixel texture. Each pixel consists of four bytes, nominally labelled r, g, b, a.

Each 32-bit pixel in our render target (destination raster) receives 2 bits from each of four channels in four distinct pixels. Thus, only 25% of the bits from the four pixels are used. The tables are constructed that all pixels are accounted for, each finding its way to another bit in another channel in another pixel.

The diffusion algorithm is described in more detail in section 2.4.1.

1.6 Confusion

This part of encryption performs two sub-steps. The first is a straight-forward XOR of the incoming data with a subkey. The second is a remapping of byte values via a 256-value lookup table. The LUT is a permutation derived from a subkey during initialization. A separate table is used for each row, resulting in 32 LUTs used per round (for a 4KB block size).

1.7 Implementation on GPU

Mumblepad was created to run on GPUs found in almost all smartphones, tablets and computers. The reference implementation runs in C++ code on a typical CPU.

Here is a comparison of the C++ implementation of the encryption diffuse step with the same code in an OpenGL fragment shader.

```
void CMumblepad::EncryptConfuse(uint32_t round)
{
    uint32_t x, y;
    uint8_t *clav;
    uint8_t *src;
    uint8_t *dst;
    uint32_t *prm;
    uint32_t numRows = mMumInfo->numRows;

    // second pass for encrypt
    // source = 1, destination = 0
    src = mPingPongBlock[1];
    dst = mPingPongBlock[0];
    clav = mMumInfo->subkeys[round];

    for ( y = 0; y < numRows; y++ )
    {
        prm = mMumInfo->permuteTables8bit[round][y];
        for ( x = 0; x < MUM_CELLS_X; x++ )
        {
            *dst++ = (uint8_t)prm[ (uint8_t)(*src++ ^ *clav++) ];
            *dst++ = (uint8_t)prm[ (uint8_t)(*src++ ^ *clav++) ];
            *dst++ = (uint8_t)prm[ (uint8_t)(*src++ ^ *clav++) ];
            *dst++ = (uint8_t)prm[ (uint8_t)(*src++ ^ *clav++) ];
        }
    }
}
```

```
precision mediump float;
varying vec2 v_texCoord;
uniform sampler2D source;
uniform sampler2D lutKey;
uniform sampler2D lutXor;
uniform sampler2D lutPermute;
void main(void)
{
    vec4 clav = texture2D(lutKey, v_texCoord);
    vec4 src  = texture2D(source, v_texCoord);
    vec4 xorKey;
    xorKey[0] = texture2D(lutXor,vec2(src.r,clav.r))[0];
    xorKey[1] = texture2D(lutXor,vec2(src.g,clav.g))[0];
    xorKey[2] = texture2D(lutXor,vec2(src.b,clav.b))[0];
    xorKey[3] = texture2D(lutXor,vec2(src.a,clav.a))[0];
    gl_FragColor[0] = texture2D(lutPermute,vec2(xorKey[0],v_texCoord[1]))[0];
    gl_FragColor[1] = texture2D(lutPermute,vec2(xorKey[1],v_texCoord[1]))[0];
    gl_FragColor[2] = texture2D(lutPermute,vec2(xorKey[2],v_texCoord[1]))[0];
    gl_FragColor[3] = texture2D(lutPermute,vec2(xorKey[3],v_texCoord[1]))[0];
}
```

2 Technical Description

This section will describe Mumblepad at a more detailed level.

2.1 Tables

This section describes the data tables used by Mumblepad, roughly sketching their generation. We will use the capitalized-bold word '**Key**' to represent the original 4096-byte key used.

2.1.1 Prime Number Table

This is a static table of 256 prime numbers, in the range of 3 to 4093. They were chosen randomly from the set of 564 prime numbers which are less than 4096. 4093 is the largest in this list, and is the 564th prime number. We exclude 2, the first prime number, from our table, as it is not relatively prime to 4096. The first eight values of the table are 2609, 3571, 2287, 3167, 499, 1087, 43, 2293. The full table is defined in the source file mumengine.cpp.

This table is used for obtaining prime numbers needed mainly for the generation of other tables. Usually, a byte value from the **Key** is used as an index, obtaining a quasi-random prime number. The index byte from **Key** is used only once in a given context. The table is static; it does not change when a new key is used.

2.1.2 Prime Cycles with Offset

A prime cycle is a intermediate structure used in the generation of subkeys. It refers to a data block which you would obtain from the original key, if you sampled every Pth byte, where P is a prime number in our table. This is done for all 4096 bytes in the key, using a modulus for the index. For example, if P is 3, we would end up with an array of 4096 bytes, using the following indices of bytes in the key: 0, 3, 6, ..., 4095, 2, 5, 8, ..., 4094, 1, 4, 5, ..., 4093. All the bytes in the key are redistributed; they are merely read from the key using a modulus.

We will vary this simple permutation of the key, by starting at an arbitrary offset into the key.

2.1.3 Subsidiary keys

We generate subsidiary keys from **Key**. These subsidiary keys are the same size as **Key.** They are deterministically generated from **Key.**

2.1.3.1 The Use of Subsidiary Keys

560 subkeys are generated during initialization. 304 are used for the Mumblepad initialization;

- one per round in the confusion pass as an XOR step (8)
- one per round, in the generation of bitmask permuations used in the diffusion pass (8)
- one per round, per each of 32 rows, in the confusion pass as a LUT (256).
- four per round, in the diffusion pass for the remapping (32).

An additional 256 subkeys are used by the pseudorandom number generator during an XOR pass, in addition to providing 256 bytes for the RC4 state initialization. In the multi-threaded implementation, every PRNG belonging to one of 16 worker threads uses its own set of 16 subkeys.

2.1.3.2 Generation

The generation of a subkey is performed in a deterministic manner using several prime cycles with offsets XORed with themselves. Indiced are implied mod 4096.

```
primeCycle(primeTable[key[i]],offset)
primeCycle(primeTable[key[i+3]],offset+5)
primeCycle(primeTable[key[i+6]],offset+10)
primeCycle(primeTable[key[i+9]],offset+15)
primeCycle(primeTable[key[i+12]],offset+20)
primeCycle(primeTable[key[i+15]],offset+25)
primeCycle(primeTable[key[i+18]],offset+30)
```

Seven different prime cycles are generated for one subkey. The 4096-byte structure are XOR'ed with one another to generate the bytes for a specific subkey. No key byte is used twice, and no offset is used twice. 560 subkeys are generated altogether, which use 3920 prime cycles. The above algorithm would generate an identical structure after 4096 iterations.

In general, a) the subkeys should be 'random' (i.e. highly entropic), b) the subkeys should be deterministically created from **Key**, and c) the creation process should be one-way.

2.1.4 Permutation Tables

In the context of this specification, a **permutation table** is a mapping of value to another. This is essentially an N-value lookup table, where N is some power-of-two. The process is reversible, as each mapped value occurs only once in the table. To be mathematically concise, it is a **bijection** or **bijective mapping**, a specific kind of isomorphism. For instance, one of the confuse encryption operations performed is a byte remapping: one 8-bit value is mapped to another value. This operation employs a 256-value permutation table. Mumblepad uses 256 of these tables, all distinct. Additionally, we generate 3-bit (8-value) and 10-bit (1024-value) tables.

2.1.4.1 Generation

This section briefly describes how the permutation tables are generated. The requirements for the tables are simply described:

-- correctly sized array, with no repeated or missing values, ergo reversible
-- deterministically derived from a single-use subkey
-- they should be 'random', i.e., mathematically indistinguishable from a true random “shuffling of the deck”.

A general algorithm, could be stated thusly for generating an N-value permutation table.

a) We instantiate a **List** of N numbers, numbered 0..N-1

b) We fetch a random number, n, in range 0...N-1. This becomes the first value in our **Order** table.

c) We remove n from **List**. **List** now contains N-1 values

d) We then fetch a random number in the range 0..N-2. This number is used as an index into List, giving us a new n. This n is added to **Order**, being the second value.

e) We remove n from **List**. **List** now contains N-2 values.

f) We repeat this process until there is just one remaining value in **List**, and **Order** contains now N-1 values. We make this last value the last value in **Order**, which is now our final table, containing N values.

This is basically the algorithm we use for Mumblepad. The trick is to use the subkey in steps b and d in a deterministic way. Here, we simple grab 32-bit integers from the subkey, in sequence, to provide deterministic "random" numbers. To get a random number then in the range of 0..N-1, you just do a modulus N on your 32-bit sample: sample % N.

In this way, all of our permutation tables are perfectly reproduceable from our **Key**, via the **Subkeys**.

2.1.4.2 Inverse Tables

Separate tables are required for decryption. They perform inverse function, and are normally genrerated in the following manner:

```
int permuteTable[256];
int inverseTable[256];
for ( n = 0; n < 256; n++)
    inverseTable[permuteTable[n]] = n;
```

2.1.5 Bitmasks

For each round, there exists a set of four bitmasks. Each bitmask is a value consisting of two bits, chosen from the eight bits within a byte; each bit appears only in one of the four bitmasks. The set of four bitmasks thus contain all 8 bits, and total 255. They will be used in the diffusion pass described later.

A set of bitmasks is generated from an 8-value permutation table. A typical table might look like this: 3, 0, 6, 4, 7, 2, 5, 1.

The four bitmasks generated would then be:

```
0x09 = 9   = 8 + 1   = (1<<3) + (1<<0)
0x50 = 80  = 64 + 16 = (1<<6) + (1<<4)
0x84 = 132 = 128 + 4 = (1<<7) + (1<<2)
0x22 = 34  = 32 + 2  = (1<<5) + (1<<1)
```

We can verify: 9 + 80 + 132 + 34 = 255. All of the bits are used.

2.2 Pseudorandom Number Generator

We pad blocks with a small number of "random" unsigned bytes. These are created by an RC4 PRNG stream. We XOR outgoing data with the assigned set of 16 subkeys, 64KB in size. A portion of the subkey data for the initialization of the RC4 state machine.

2.3 Input Block Structure

The data which gets sent to Mumblepad for encryption is structured as blocks. Each block contains plaintext, a small number random bytes generated by the pseudorandom number generating, and some metadata.

The metadata is 8 bytes long, and consists of a 32-bit checksum, a 16-bit length, and a 16-bit sequence number. The number of random bytes varies with the block size.

Block size	Plaintext	Random	Metadata
4096	4000	88	8
2048	2000	40	8
1024	1000	16	8
512	492	12	8
256	240	8	8
128	112	8	8

The Mumblepad block cipher does not use any block cipher mode in the classical sense. There is no forward feedback used, no initialization vector, no salt. However, this does not mean that blocks encryped with the same payload result in the same ciphertext. On average, an optimal 255/256 of the bytes do change, block-to-block using the same plaintext. This is due to the extensive diffusion performed in the first pass in combination with random padding bytes. This will be explained next.

2.3.1 Plaintext Packing

Here we sketch how raw plaintext is packed into the power-of-two-sized data chunks. The block structure consists of 9 fields: 2 plaintext data chunks, 4 padding chunks, a checksum, a length, a sequence number (see also source file mumdefines.h)

paddingA	2	2	2	4	16	32
dataA	72	148	304	618	1236	2472
paddingB	2	2	4	4	4	12
checksum	4	4	4	4	4	4
length	2	2	2	2	2	2
seqnum	2	2	2	2	2	2
paddingC	2	2	4	4	4	12
dataB	40	92	188	382	764	1528
paddingD	2	2	2	4	16	32
total	128	256	512	1024	2048	4096

2.4 Cipher Round

Now we can finally talk about the encryption process itself. This consists of eight rounds on one

block of packed plaintext. We will use the capitalized-bold word '**Plaintext**' to refer to the 4096-byte block passed to the encyption engine.

One round of Mumblepad consists of two passes: diffusion and confusion. The former moves bits around in within the data block, preserving their values, while the latter changes the bit values themselves. When encrypting, diffusion is always performed first. On decryption, the order of the passes is reversed: inverse confusion is followed by the inverse diffusion operation.

Each pass uses two main pieces of memory, both corresponding to the block size: a source block and a destination block. A given pass will use the source block, along with the proper key and other tables, and write data to the destination block. This will then be uses as the source block on the next pass. One round consists of an A-to-B operation, followed by a B-to-A pass, operating in a ping-pong fashion.

At the start of encryption, **Plaintext** is written to chunk A. There are 16 passes in all, two passes for each of eight rounds. The final encrypted data ends up in chunk A. When doing decryption, the original ciphertext is written to chunk A.

2.4.1 Diffusion

The 4KB-block is organized as a 32x32 raster of cells, where each of the 1024 cells is a 32-bit value. We further divide the 32-bit cell into four 8-bit **channels**. These would correspond to the four color channels which make up a 32-bit pixel. Each cell therefore has an x and y coordinate, which we will sometimes refer to as row and column. These values range from 0 to 31.

Each pixel in our destination 32x32 grid points to a set of four source pixel positions. Two bits are fetched from each channel of each pixel. Eight bits are grabbed from each pixel, 32 in total for the render target pixel.

```
unsigned char p1[4];
unsigned char p2[4];
unsigned char p3[4];
unsigned char p4[4];
dst[0] = (p1[0] & maskA) + (p2[2] & maskB) + (p3[3] & maskC) + (p4[1] & maskD);
dst[1] = (p1[2] & maskA) + (p2[3] & maskB) + (p3[1] & maskC) + (p4[0] & maskD);
dst[2] = (p1[3] & maskA) + (p2[1] & maskB) + (p3[0] & maskC) + (p4[2] & maskD);
dst[3] = (p1[1] & maskA) + (p2[0] & maskB) + (p3[2] & maskC) + (p4[3] & maskD);
```

```
unsigned char p1[4];
unsigned char p2[4];
unsigned char p3[4];
unsigned char p4[4];
destination channel 0 =
      source 0, channel 0 AND'ed with bitmask 0 +
      source 1, channel 2 AND'ed with bitmask 1 +
      source 2, channel 3 AND'ed with bitmask 2 +
      source 3, channel 1 AND'ed with bitmask 3 +
destination channel 1 =
      source 0, channel 2 AND'ed with bitmask 0 +
      source 1, channel 3 AND'ed with bitmask 1 +
      source 2, channel 1 AND'ed with bitmask 2 +
      source 3, channel 0 AND'ed with bitmask 3 +
destination channel 2 =
```

```
      source 0, channel 3 AND'ed with bitmask 0 +
      source 1, channel 1 AND'ed with bitmask 1 +
      source 2, channel 0 AND'ed with bitmask 2 +
      source 3, channel 2 AND'ed with bitmask 3 +
destination channel 3 =
      source 0, channel 1 AND'ed with bitmask 0 +
      source 1, channel 0 AND'ed with bitmask 1 +
      source 2, channel 2 AND'ed with bitmask 2 +
      source 3, channel 3 AND'ed with bitmask 3 +

dst[0] = (p1[0] & maskA) + (p2[2] & maskB) + (p3[3] & maskC) + (p4[1] & maskD);
dst[1] = (p1[2] & maskA) + (p2[3] & maskB) + (p3[1] & maskC) + (p4[0] & maskD);
dst[2] = (p1[3] & maskA) + (p2[1] & maskB) + (p3[0] & maskC) + (p4[2] & maskD);
dst[3] = (p1[1] & maskA) + (p2[0] & maskB) + (p3[2] & maskC) + (p4[3] & maskD);
```

Diffusion is complex, doing many things at once. It is a combination of position mapping, channel mixing, bit masking and addition.

2.4.2 Confusion

Confusion is a relatively straight-forward operation. First, a bitwise XOR is performed with the subkey used for the particular round and purpose. Each of the 4096 bytes in the source is XOR'ed with its corresponding byte in the **Subkey**.

Secondly, the resulting byte is replaced with a corresponding byte in a 256-value remapping (permutation) table. Not only do these tables change for each round, but a different table is used for each row during a single pass. There are then 32 tables for each of eight rounds, resulting in 256 tables altogether. This is equivalent to the S-box in the AES cipher algorithm (the SubBytes, or substitute bytes function). Although only one mapping is used by AES, our table consists of 256 different mappings, for a total of 65536 bytes.

2.5 Decryption

When performing decryption, the entire process is reversed. Per-round subkeys go in reverse order, from 8 to 1. Inverse confusion is followed by inverse diffusion. A set of inverse tables are used. The performance is exactly the same as that for encryption, regardless of the engine implementation used.

3 Library API

This section describes the public API, which is found in the mumpublic.h header file.

```
void * MumCreateEngine(EMumEngineType engineType, EMumBlockType blockType,
   EMumPaddingType paddingType, uint32_t numThreads);
```

This function instantiates an engine. There are four engine types right now, two CPU implementations, and two GPU implementations.

MUM_ENGINE_TYPE_CPU: This is the single-threaded CPU implementation.
MUM_ENGINE_TYPE_CPU_MT: This is the multi-threaded CPU implementation
MUM_ENGINE_TYPE_GPU_A: This is a single block OpenGL implementation
MUM_ENGINE_TYPE_GPU_B: This is a multi-block OpenGL implementation

The block type is merely one of the six sizes:

MUM_BLOCKTYPE_128
MUM_BLOCKTYPE_256
MUM_BLOCKTYPE_512
MUM_BLOCKTYPE_1024
MUM_BLOCKTYPE_2048
MUM_BLOCKTYPE_4096

The padding type is there for analytical and testing purposes. It makes it possible to disable the padding mechanism, and send plaintext to the encryption engine which is the same size as the encryption block. Of course, when this is done, there is no checksum, sequence numbering, and no length stored. Most importantly, when the same plaintext is encrypted twice, the encrypted data will be the same.

Lastly, the number of threads may be specified. This parameter is only relevant for the multi-threaded CPU implementation; for all others it is ignored.

```
EMumError MumInitKey(void *me, uint8_t *key)
EMumError MumLoadKey(void *me, char *keyfile)
```

These two functions initialize the engine with a 4096-byte key. The key may be passed in via an unsigned char pointer, or it may be loaded via a file. Any calls to decryption or encryption methods will fail before the engine is initialized with a key.

```
void MumDestroyEngine(void *me)
```

This function deletes the engine, releasing all resources.

```
EMumError MumGetSubkey(void *me, uint32_t index, uint8_t *subkey)
```

This function is purely for analytical purposes. It returns one of the 560 subkeys.

```
EMumError MumPlaintextBlockSize(void *me, uint32_t *plaintextBlockSize)
```

This function returns the maximum size of plaintext in a block. This value will be one of the following: 112, 240, 492, 1000, 2000, 4000. If padding is turned off in the engine (for testing and/or analysis), the full block sizes are returned here.

```
EMumError MumEncryptedBlockSize(void *me, uint32_t *encryptedBlockSize)
```

This function returns the size of the encrypted block. It will be one of the following: 128, 256, 512, 1024, 2048, 4096.

```
EMumError MumEncryptedSize(void *me, uint32_t plaintextSize, uint32_t
   *encryptedSize)
```

This function computes the size of an encrypted block of memory, given the size of the plaintext.

```
EMumError MumEncryptBlock(void *me, uint8_t *src, uint8_t *dst, uint32_t length,
   uint32_t seqnum)
```

This function encrypts a single block of plaintext, returning the excrypted data in the destination pointer. The length must not be greater than the plaintext block size for the particular encryption engine.

```
EMumError MumDecryptBlock(void *me, uint8_t *src, uint8_t *dst, uint32_t
   *length, uint32_t *seqnum)
```

This function decrypts a single block of encrypted, returning the decrypted data in the destination pointer. The length of the plaintext is returned, as well as the 16-bit sequence number.

```
EMumError MumEncrypt(void *me, uint8_t *src, uint8_t *dst, uint32_t length,
   uint32_t *outlength, uint16_t seqNum)
```

This function encrypt a unit of memory which may consist of many blocks.

```
EMumError MumDecrypt(void *me, uint8_t *src, uint8_t *dst, uint32_t length,
   uint32_t *outlength)
```

This function decrypts of a region of encrypted data. The length must be a multiple of the encrypted block size.

```
EMumError MumEncryptFile(void *me, char *srcfile, char *dstfile)
```

This function encrypts the source file, writing the encrypted version to the destination file name

```
EMumError MumDecryptFile(void *me, char *srcfile, char *dstfile)
```

This function decrypts a file, writing the original plaintext version to the destination file name.

```
EMumError MumUtilCreateEncryptedFileName(EMumBlockType blockType, char
   *infilename, char *outfilename)
```

This function created a filename for an encrypted file. Although not strictly required by the engine's API, as a convention it would be useful to name encrypted files in a standard way. We do this by appending a file extenstion to the name of the original file. The file extension is one of 6, based on the encryption block size.

```
128 bytes  = .mu1
256 bytes  = .mu2
512 bytes  = .mu3
1024 bytes = .mu4
2048 bytes = .mu5
4096 bytes = .mu6
```

For example, if the block size is 1024 bytes, and the file is called readme.txt, the encrypted file would be called readme.txt.mu4. If the files is called installer.exe, and the block size is 4096, then the encrypted file would be named installer.exe.mu6.

This enables restoration of the original file name when decrypting it later. Engines using the same key with different block sizes are not compatible with each other.

```
EMumError MumUtilGetInfoFromEncryptedFileName(char *infilename, EmumBlockType
   *blockType, char *outfilename)
```

This function essentially does the reverse of the previous function. Given an encrypted file name, it returns the original name (stripped of the our file extension), along with the block type.

N.B.: there are no restrictions on filenames used with the EncryptFile and DecryptFile functions. You can choose to have the same names but in different directories. It may not be necessary for you store the engine block size in the filename.

4 Mumblepad Library

The Mumblepad software is provided as static libraries for Windows. They are found in the \lib directory. The two headers files required for a client application are located in \includes. The \mumblepad directory contains the full source code for the libraries, as well as Visual Studio Express 2015 project and solution.

Eleven files (4 .cpp files and 7 header files) form the core of Mumblepad, providing the reference specification of the code (pending a full-on technical specification document).

mumdefines.h
mumrenderer.h
mumtypes.h
mumblepad.cpp/h
mumengine.cpp/h
mumprng.cpp/h
mumpublic.cpp/h

4.1 GPU Implementation

As mentioned above, the library included four different implementations of the encryption engine. They all are considered 'renderers', and indeed, they implement a CMumRenderer C++ interface.

The two GPU implementations are found in mumblepadgla.cpp/h, and mumblepadglb.cpp/h. The GPU-A implements a single block, performed in a 32x32 square. This size may be smaller, based on the block size. There may be 16 rows (2048 bytes), 8 rows (1024 bytes), 4 rows (512 bytes), 2 rows (256 bytes), or 1 row (128 bytes).

The second GPU implementation operates on 8 rounds at a time, using 32-rows per block: only the 4096-byte size is supported in this implementation. Important to note, is that there is a certain latency when using this implementation. Eight blocks are in process at once, in a texture 32x256. Each 32x32 block passes its results to the block underneath. New blocks are added at the top, and finished data ends up on the top after the eighth round.

5 Mumblepad Demo/Test Project

In the mumbletest directory there is a Visual Studio project which links to the Mumblepad library. It runs a test suite (mumbletest.exe) and profiles the four different Mumblepad renderers. It also may be configured to generate reference files.

The file main.cpp shows how to create an engine, and how to encrypt/decrypt individual blocks, large memory areas, and whole files. The profiling uses a 256MB memory segment.

5.1 Test files

For file testing and profiling, I have chosen two artibrary files to use: a JPEG and an MPEG-4. They are fairly large (greater than 10MB), but less than 50MB.

testfiles\image.jpg is from here:
http://upload.wikimedia.org/wikipedia/commons/b/be/Titian_Bacchus_and_Ariadne.jpg

The first file is a large jpeg file, from Wikipedia. It is an image of Titian's “Bacchus and Ariadne”; the JPEG file is 4670x4226 pixels

testfiles\music.m4a is from here:
http://www.blockmrecords.org/bach/audio/aac/256/BWV0565.m4a

The second file is a recording of James Kibble playing Bach's Toccata and Fugue in D minor.

5.2 Reference files

The reference files consist of two original plaintext files: a smaller image.jpg file, and constitution.pdf. There is a key file, and a set of six encrypted files for each of the two plaintext files. The test suite tests the decryption of the files using the appropriate block size for the engine used.

The test/demo program also has a compile flag which enables generation of the reference files, and writing out the accompanying key file.